Universidade de Aveiro
2017

Departamento de Eletrónica,
Telecomunicações e Informática

RAFAEL
FERREIRA

Redes Programáveis para Vídeo

Programmable Networks for Video

Universidade de Aveiro
2017

Departamento de Eletrónica,
Telecomunicações e Informática

RAFAEL
FERREIRA

Redes Programáveis para Vídeo

Programmable Networks for Video

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Rui Luís Andrade Aguiar, Professor catedrático do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Daniel Corujo co-orientador, Professor adjunto convidado da Escola Superior de Tecnologia e Gestão de Águeda da Universidade de Aveiro.

Dedico este trabalho à minha namorada e família pelo incansável apoio.

o júri / the jury

presidente / president	Prof. Doutor Andre Ventura da Cruz Marnoto Zuquete Professor Auxiliar, Universidade de Aveiro
vogais / examiners committee	Prof. Doutor Paulo Manuel Martins Carvalho Professor Associado, Escola de Engenharia - Universidade do Minho
	Prof. Doutor Rui Luís Andrade Aguiar Professor Catedratico, Universidade de Aveiro

Palavras Chave redes definidas por software, vídeo, uplink, live stream, redes móveis.

Resumo Com o crescimento das redes sociais a criação de conteúdo por parte do utilizador começou a ser muito virada para o vídeo, inspirado pelo sucesso do Youtube. Redes sociais como o Instagram, Facebook, Snapchat e Twitter permitem que os utilizadores gravem vídeos em direto, histórias em vídeo ou imagens de alta resolução. Isto acarreta custos para o operador e preocupações para as plataformas sociais. A forma como a rede de um operador se comporta influencia muito a experiência do utilizador ao utilizar este tipo de serviços. Esta dissertação apresenta uma solução para que o prestador de serviço consiga ajustar a rede do operador aos seus conteúdos e garantir uma boa qualidade de serviço. Para isso propõe o uso de Software-Defined Networks criando um mecanismo em que várias entidades possam influenciar a operação da rede em prol dos conteúdos gerados em vídeo. Este mecanismo consiste num intermediário entre a rede e outras entidades além destes, nomeadamente uma aplicação de monitorização da rede, prestadores de serviço e até o próprio operador. Também é apresentada para esta solução uma visão genérica de como é possível usar este conceito e arquitetura em diferentes tipos de cenários, decompondo os componentes e mantendo as suas funcionalidades. Esta solução é validada com a implementação de duas provas de conceito, com arquiteturas semelhantes mas sendo uma delas otimizada para dispositivos de rede e servidores de baixa capacidade. Por fim, são apresentados resultados e conclusões para as duas implementações desenvolvidas.

Keywords Software defined networks, video, uplink, live stream, mobile networks.

Abstract

With the growth of social networking, the content creation by the user started to be very video-oriented. Inspired by the success of Youtube, social networks like Instagram, Facebook, Snapchat and Twitter Periscope allow users to record live videos, stories or high resolution images. This entails costs for the operator and concerns for social platforms. The behavior of an operator's network greatly influences the user experience when using this type of service. This dissertation presents a solution for the service provider to be able to adjust the operator's network to its contents and to guarantee a good quality of service. For this it proposes the use of Software-Defined Networks creating a mechanism in which several entities can influence the network in Content generated video. This mechanism consists of an intermediary between the network and other entities, namely a network monitoring application, service providers and even the operator. This solution also presents a generic view of how it is possible to use this concept and architecture in different types of scenarios, decomposing the components and maintaining their functionalities. This solution is validated with the implementation of two proofs of concept, with similar architectures but with one of them optimized for network devices with low capacity. Finally, results and conclusions are presented for the two implementations developed.

Contents

List of Figures

List of Tables

Glossary

3GPP	3rd Generation Partnership Project
APP	Application
AP	Access Point
API	Application programming interface
ARM	Advanced RISC Machine
AVC	Advanced Video Coding
BSR	Buffer Status Report
CFP	Contention Free Period
CLI	Command-line Interface
CQI	Channel Quality Indicator
CSMA/CA	Carrier Sense Multiple Access – Collision Avoidance
CTS	Clear to Send
CU	Coding Unit
DCF	Distributed Coordination Function
DHCP	Dynamic Host Configuration Protocol
EDCA	Enhanced Distributed Channel Access
EPS	Evolved Packet System
FLV	Flash Video
Gb	Gigabyte
GBR	Guaranteed Bit Rate
HCCA	HCF Controller Channel Access
HCF	Hybrid Coordination Function
HEVC	High Efficiency Video Coding
HLS	HTTP Live Streaming
HTTP	Hypertext Transfer Protocol
IDR	Instantaneous Decoding Refresh
IP	Internet Protocol Address
It	Intituto de Telecomunicações - Polo Aveiro
JVT	Joint Video Team
L2	Layer 2
LCU	Largest Coding Unit
LTE	Long Term Evolution
MP3	Moving Picture Experts Group Layer-3 Audio
MCS	Modulation and Coding Scheme
MME	Mobility Management Entity
MUGV	Mobile User Generated Video
NAL	Network Abstraction Layer
NFV	Network Function Virtualization
ONF	Open Networking Foundation
OS	Operating System
OTT	Over-the-top
OVS	Open vSwitch
P-GW	Packet Data Network Gateway
PCF	Point Coordination Function
PoC	Proof of Concept
QoS	Quality of Service
RTCP	RTP Control Protocol
RTMP	Real-Time Messaging Protocol
RTP	Real-Time Transport Protocol
RTS	Request to Send
RTSP	Real Time Streaming Protocol
S-GW	Serving Gateway
SDN	Software Defined Networking
SEI	Supplemental Enhancement Information
SLA	Service-Level Agreement
TC	Traffic Categories
TCMA	Tiered Contention Multiple Access
TCP	Transmission Control Protocol
TFT	Traffic Flow Template
UDP	User Datagram Protocol
UE	User Equipment
UHD	Ultra-high-definition
VCL	Video Coding Layer
VSA	Video SDN Application
VM	Virtual Machines
WANs	Wide Area Networks

CHAPTER 1

Introduction

Social Network Services sucha as Twitter is Periscope, Facebook Live or Instagram Live, allow users to do live transmission, i.e., broadcast from their mobile devices for their followers or friends.

These services are expected to become mainstream during the upcoming years, translating into huge amounts of video data. Some users of this kind of service upload their life to the Internet for the followers to be entertained, motivated, and maybe an endless amount of reasons. But, when these services become mainstream, such as users that went to events with a lot of people and upload it to the social networks such as Instagram with the live service, this turns into a huge problem for network operators, meaning that they will probably face a bottleneck in the network.

This work addresses these issues and provides a platform that leverages SDN technologies to improve traffic interaction preferably between the service providers and the network operators.

1.1 Motivation

Due to the increase of the use of social mobile applications, users tend to use mobile networks to reach their social networks. Nowadays, social networks have added features like video sharing and live video streaming to users and their followers. These features gain from day to day more popularity and users.

Network operators face a new problem related with mobile networks because the users are changing the way that they use the mobile networks. They are sharing more video and photo content, given the bandwidth and requirements needed for such type of contents they will need to continuously upgrade their mobile networks.

Given the increased interest in Software Defined Networkings (SDNs) technologies, it is the motivation of the present work to define a platform for Mobile User Generated Video (MUGV) services while leveraging SDN technologies.

VDSNet is a project funded by Altran[1], an international group which offers innovation and high-tech engineering consulting, and developed in a partnership between Altran Research and Intituto de Telecomunicações - Polo Aveiro (It).

1.2 Objectives

The purpose of the present work is the prioritization of uplink bandwidth providing a platform to service providers (i.e. Facebook), network operators and other network entities to request more bandwidth for their services (i.e. Video stream). Thus, this work main objective is responsible for developing a platform that makes use of SDN technologies and provides to the stakeholders a way of requesting more bandwidth or priority for theirs services through an Application programming interface (API) that will be the same independently the location that will be deployed. Providing such API will bring more abstraction to the service provider, which is useful.

At the end, a Proof of Concept (PoC) demonstrates the abstraction created and will be demonstrated with different types of audiences.

Additionally, this document intends to provide a description of all implementation steps that were taken.

1.3 Contributions

This work contributes to the project mentioned before, which intends to optimize the usage of mobile network transmitting video with this work able to make a PoC for that purpose.

The work that was made, was presented for four schools in Intituto de Telecomunicações - Polo Aveiro and at Altran in Lisbon, in demonstration events to the public and in VDSNet project meetings, respectively.

With the development of this work, there are some contribution with the open source project in GitHub[2], with some modifications that had to be done in order to work with Raspberry Pi 3.

1.4 Structure

The following Chapter 2 intends to create a familiarization and background with the most important concepts that will be used in this document, inside that chapter, both conceptually and technically. Afterwards, Chapter 3 describes the work problems that are addressed and the final use cases. That chapter intends to give more abstract and conceptual background, presenting how that problem can be solved using the solution described. With the previous chapter and State of the Art background, Chapter 4 provides a look at the implementation in two steps, first describing how the used technologies helped to develop a PoC and then describing the two PoC is that were made. With the implementation addressed, tests are

[1] `http://www.altran.com/`
[2] `https://github.com/containernet/containernet`

carried out and results measured and discussed, as presented in Chapter 5. Finally, a conclusion is provided in the last chapter, Chapter 6, where possible future work is also discussed. Appendices and References are presented afterwards.

CHAPTER 2

State of the Art

In this chapter will be introduced the State of the Art, some concepts and technologies that are important for the development of the work.

2.1 Virtualization

Virtualization[1] provides a simple way of dividing resources of a machine into multiple virtual environments. In the beginning[2], virtualization was the ability of dividing computing power in large computing systems. Then, it was possible to run different versions of the same operating system in each one of the partitions created. That ability allows a single computer to play the role of several computers, through sharing the hardware power for the multiple environments created.

There are some different types of virtualization[3] nowadays, such as:

- Network
- Storage
- Application
- Hardware
- Desktop

Some of them will be discussed with more detail later in this dissertation.

2.1.1 Operating system virtualization

A hypervisor is responsible for the control of multiple virtual machines in a single physical machine using virtualization technologies[4].

Operating system virtualization[5] means that the same operating system will be shared by all the applications and we can share the hardware resources between all the applications or we can create partitions.

If the applications that are using this type of virtualization only need to make simple and user calls to the operating system, it means that we are saving hardware resources consumption because there is only one operating system instantiated.

Hypervisors tend to put more effort in isolated applications than the creation of sharing points between applications. Most of them, only allow sharing between applications through a network.

Moreover, there are some disadvantages of this kind of virtualization, as it tends to have problems with stability (when the operating system causes some failure becomes a single point of failure for all the applications) and isolation, since the operating system is the same, the isolation provided by the hypervisor tends to be different.

a) Docker

For example, Docker[6] (which has been used in this work), nowadays is known to be a tool to solve many issues to different kind of requirements.

For developers it can help when collaborating with other co-workers. The need for creating an environment shared between developers for testing, building and running code can be solved with Docker. The team leader or a responsible have to ensure that every team member that is working in the same environment can create a Docker image with all the variables, tools, libraries and binaries necessary for work and then share with others. Using Docker Registry is really helpful in that scenario because other co-workers can just pull from the repository and work.

For network operators the usage of Docker can be related with the need for running and managing applications side-by-side in isolated containers and get better computer resources utilization since Docker is a hypervisor based in Operating System virtualization. Also, enterprises are using Docker to build and make new features in production faster and more securely.

Docker website describes a container[7] as a lightweight, standalone, executable package that includes everything that one software needs to run, such as: code, runtime, system tools, system libraries and settings. Using a container means that the software will be isolated from its surroundings.

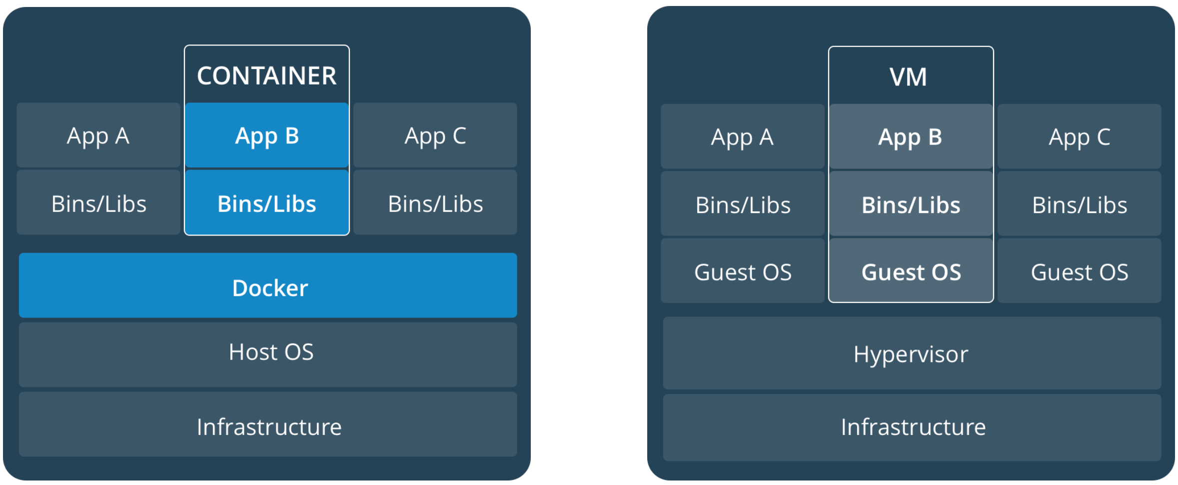

Figure 2.1: Comparing containers and virtual machines.[6]

As described above, containers are an abstraction at the Application (APP) layer, it means that multiple containers can run on the same machine and then share the same Operating System (OS) kernel with other containers. Each container is running as an isolated process in user space. So, if a container share the same OS Kernel with other containers it means that containers take less space than Virtual Machines (VM)s and start almost instantly.

Whit the abstraction at the application layer, it means that Docker provides to the container application the necessary resources to run separately from other containers. One container is stateless and immutable, if we want to backup data that is used by the container we create a Docker volume.

Virtual machines, as we can see in the image above, need a full copy of an operating system. So if we have to run a simple application we will have tens of Gigabyte (Gb)s replicated in the same single machine.

2.2 Software-Defined Networking

In current data computer networks it is possible to identify three different planes: data, control and management.

As we know, the switches forward traffic based on known internal rules only and it is not possible to get rules from outside entities.

SDN provides a new way of decoupling the data and control plane of data networks, turning the network into a programmable entity and making the devices much more simpler.

Turning a network into a programmable network shows many advantages to enterprises, network operators and administrators. They can have more and agile ways of change and interact with the network, in a browser or other programmability way.[8]

2.2.1 SDN Controller and control plane

Software Defined Networking also allows the creation of a management and dedicated entity in which the equipment can be controlled through a SDN Controller.

The SDN Controller is responsible for providing a way of controlling, orchestrating and managing the network using the control plane and creating abstractions of the network.

Uses a standardized interface, Southbound Interface, to arbitrate the control of network resources and provides a Northbound Interface for applications to automate the operation of abstracted network resources.

So, using the SDN Controller, the network operator or enterprise can make topology changes more easily using a programmable entity and making a secured and manageable configuration through their custom made applications.

It can be seen as a distributed database that keeps track of everything and changing it means that the database will be triggered and automatically configured, since the control plane is separated from the data plane, with the forwarding rules becoming directly known to the control plane.

Using that analogy it is possible to see a new and faster way to optimize, configure and make security policy rules for the enterprise or operator network.

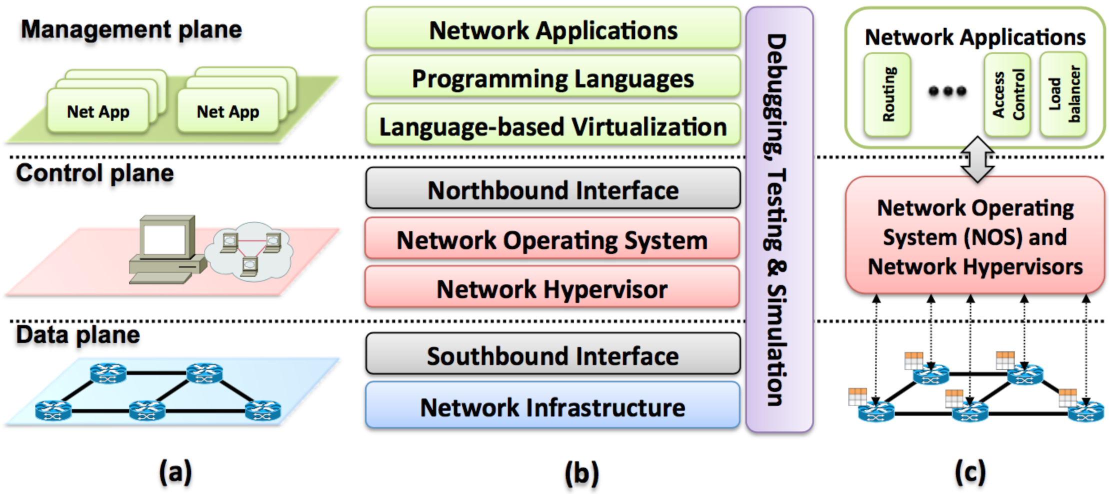

Figure 2.2: SDN Architecture[9]

b) OpenDayLight

One of the most used SDN Controllers is OpenDayLight, this SDN Controller is hosted by The Linux Foundation and is a community-driven initiative and backed by platinum members (big companies like Intel, Cisco, RedHat), gold and silver members. This initiative intends to integrate different technologies in a single platform.

2.2.2 OpenFlow

As described above, the control plane and forwarding plane need to talk and exchange information to make the forwarding decisions. A communication protocol that is responsible to carry that communication between the two elements was defined for this.

The OpenFlow[10] protocol has been created by the Open Networking Foundation (ONF) which is an organization responsible to promote the adoption of SDN and is the open standard protocol for that type of communications.

Flow[11] is a sequence of packets sent from a particular source to a particular unicast, anycast, or multicast destination.

Software Defined Networking switches consist of a group table, one or multiple flow tables and an OpenFlow-based interaction with the SDN Controller. The flow tables are composed of flow entries, which reflect the way all packets from a flow must be handled.

Flow entries match packets in a priority order, and when a match is found the instructions associated with the match (flow entry) are executed. If there is no match in the current flow table, the packet ca not be either sent to destination, then can be sent to the next flow table, dropped or forwarded using a table-miss.

c) Packet processing

An OpenFlow switch should contain at least one flow table. The flow tables are numbered from 0 to n. When a packet arrives at a switch, as described in the Figure 2.3, it must be processed by the first flow table 0 and one criteria will be applied to the packet using the packet header, leading to a "match" or "miss" event.

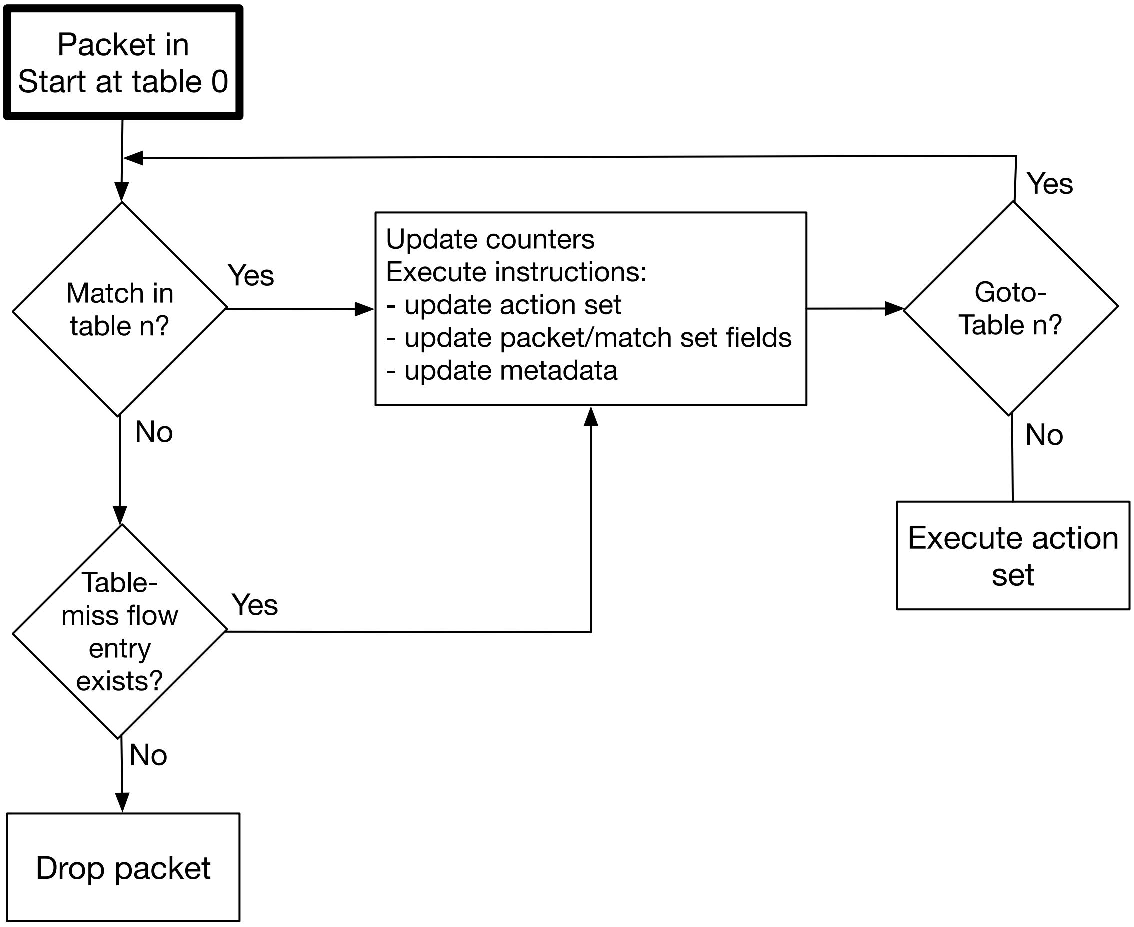

Figure 2.3: Packet processing[12]

Each flow entry has[12] instructions attached and match conditions. The instructions can be used to modify the packets state, forward the packet to a particular port, forward the packet to another flow table or group for further processing and pass some metadata in the process.

If there is a go to table action, the flow table number must be higher than the current one. Otherwise an action set will be applied to the flow.

Actions can discard, modify, queue or forward the packet. In the OpenFlow[12] version 1.0 the action set is modified directly by action lists and, in subsequent versions of the protocol, the action list is modified by the instruction structure.

There are different versions of OpenFlow protocol and each one specifies a different way of instructions that can be or not applied and which actions are supported.

2.2.3 Open vSwitch

As previously mentioned, OpenFlow switches implement exclusively the data plane functions, acting as a traffic forwarding device. Open vSwitch[13] was designed to be used in servers and is the most commonly used open source virtualized SDN switch. It is capable of forwarding traffic between VMs in the same physical machine or between the VM and the physical network. It supports standard management interfaces (e.g. sFlow[14], NetFlow[15], CLI) and enables programmatic extensions and control of its forwarding functions.

2.2.4 Mininet

Mininet[16] is an emulator of an OpenFlow network on a single machine, making it a popular tool both for researchers and developers of new services, due to the significant simplification in the initial development and deployment efforts.

It has an API written in Python which allows to interact with every component of Mininet easily, it includes the experimentation with new services or modification of protocols prior to deployment in physical infrastructures.

Mininet applies Open vSwitch as the default OpenFlow switch and we can select which OpenFlow controller we want to use.

2.2.5 Containernet

Containernet[17] is a Mininet open-source project[18], based on Mininet that allows to use Docker containers as hosts in emulated networks. This enables interesting functionalities to build networking/cloud testbeds, namely:

- Add and remove docker containers in Mininet topologies
- Use docker containers it is possible to isolate applications with different IP is of the host IP
- Execute commands inside Docker containers using the Mininet Command-line Interface (CLI)
- Add hosts or Docker containers to a running mininet topology
- Usage of docker resources sharing quota features

2.2.6 SDN for Mobile Networks

In [19], the authors highlight the SDN's potential to enable the end-to-end unification of the control plane across wireless access and mobile core network. As such, SDN is leveraged for dynamically controlling network traffic over Wide Area Networks (WANs) according to dynamic network conditions and application types.

The SDN Application is situated in edge nodes of wireless networks (e.g. Packet Data Network Gateway (P-GW) in LTE), in order to be fed by flow information such as Client ID, APP type, Priority and Quality of Service (QoS) requirements, and is responsible for enforcing QoS through multiple SDN controllers. The solution is depicted in Figure 2.4.

The authors in [20] propose the adaptation of 3GPP's Long Term Evolution (LTE) architecture so that MME is extended with SDN Controller functions, and P-GW, S-GW and eNodeBs act as SDN switches.

In [21] the concept of Connectivity Management as a Service (CMaaS) is introduced, unifying handover, mobility and routing management. The authors advocate an all-SDN network architecture for 5G, through an internet worked hierarchy of SDN controllers.

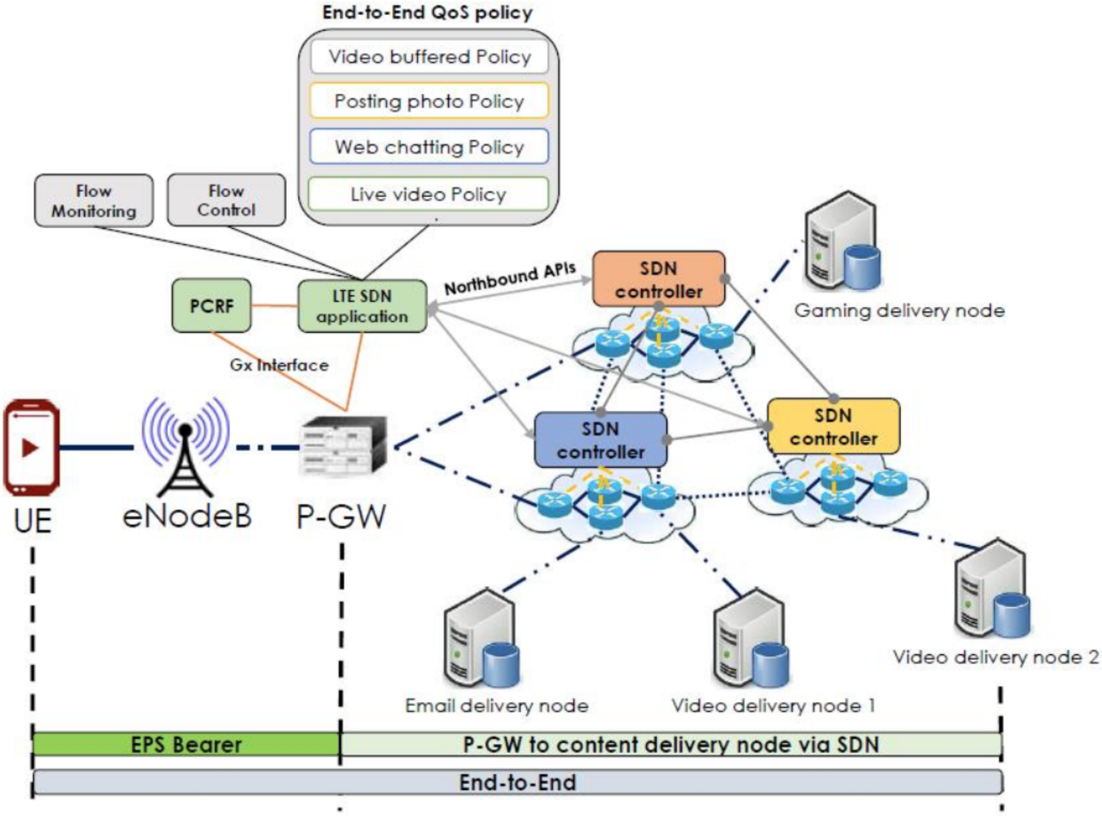

Figure 2.4: Intelligent Content Delivery over Wireless via SDN[19]

2.3 Quality of Service

Since QoS in LTE and WLAN architectures it is a relevant topic to this work and one of the use cases, this section intends to introduce some concepts that helped to develop the work architecture.

2.3.1 LTE Access

LTE provides a comprehensive QoS, policy control framework and charging system for enhancing service quality according to user-profile and application service type. QoS is implemented between User Equipment (UE) and Packet Data Network Gateway and applied to an Evolved Packet System (EPS) bearer, which is a set of network configurations to provide special treatment to a set of traffic.

There are two types of Bearers, Dedicated Bearer and Default Bearer. At least one of the former is established when UE attaches to LTE network. When a subscriber requests QoS-enabled services, a Traffic Flow Template (TFT) including a set of QoS policies is applied to the Dedicated EPS bearer (between the UE and P-GW) for policy and charging control enforcement. The QoS parameters characterizing Bearers in LTE include (Figure 2.5):

- For all bearers: QoS Class Identified (QCI)

- For Guaranteed Bit Rate (GBR) bearers: Guaranteed Bitrate (GBR) and Maximum Bitrate (MBR)
- For non-GBR bearers: Access Point Name Aggregate Maximum Bitrate (APN-AMBR) and User Equipment Aggregate Maximum Bitrate (UE-AMBR)

2.3.2 Scheduling in LTE

In LTE, eNodeB manages when UEs should send or receive data, according to information regarding the link status, the UE buffer and its flows QoS requirements. In summary, scheduling works as follows[22]:

- UE calculates the Channel Quality Indicator (CQI) pertaining to downlink channel and sends it to the eNodeB;
- UE transmits its Buffer Status Report (BSR) to eNodeB
- eNodeB computes Modulation and Coding Scheme (MCS) and Physical Resource

Block (PRB) based on the received information, and sends them to the UE through downlink channel.

As such, Scheduling in LTE is mostly based on how the radio access is perceived by the network operator (eNB), but still taking into account UE traffic load.

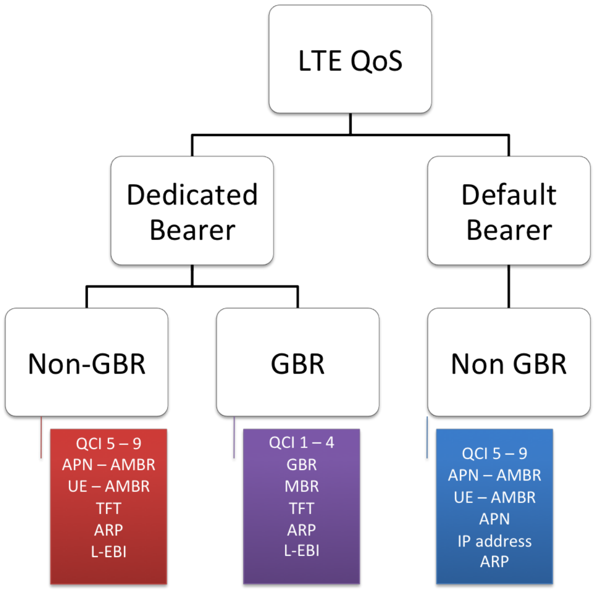

Figure 2.5: LTE QoS parameters and bearers types[23]

2.3.3 LTE QoS

GBR bearer allows to differentiate the allocation of traffic according to the Quality of Service. Different types of traffic requires different types of QoS.

Most of OTT video streaming applications and services do not transmit data continuously, and using a GBR bearer would waste network resources. Thus, in time-unconstrained video upload, using non-GBR is better suited for the stream scenario.

However, making a live streaming, a GBR bearer would be more suitable because the priority matters. Nevertheless, that the QoS concept adopted in LTE, being class-based, has no regards whatsoever on the specificities of either up-link or Ultra-high-definition (UHD) video.

2.3.4 WLAN QoS

IEEE 802.11e amendment defines a set of QoS enhancements for WLAN applications, and was incorporated in IEEE 802.11-2007 standard.

The original IEEE 802.11 standard used two modes:

- Distributed Coordination Function (DCF): in this mode the multiple stations compete for the medium and rely on Carrier Sense Multiple Access – Collision Avoidance (CSMA/CA) and Request to Send (RTS)/ Clear to Send (CTS) mechanisms. It does not allow traffic differentiation.
- Point Coordination Function (PCF): in this mode, mobile stations are coordinated by an Access Point (AP) – similarly to the role of eNB in LTE networks. PCF mode is split in three phases: Beacon transmission, Contention Free Period (CFP) and CP. In Beacon stage, the AP is responsible for sending beacon frames; in CFP, AP informs each user of the right to transmit a packet, while in CP, DCF is used.

802.11e enhances both modes of the original 802.11 standard. IEEE 802.11ac by introducing a new coordination function, Hybrid Coordination Function (HCF), and by adding Traffic Categories (TC), which assign different priorities (Access Categories - ACs) to traffic. The two proposed modes, based on DCF and PCF, are Enhanced Distributed Channel Access (EDCA) and HCF Controller Channel Access (HCCA), respectively.

EDCA introduces Tiered Contention Multiple Access (TCMA), which improves CSMA/CA, and enables for higher-priority traffic to wait less than low-priority traffic to be transmitted. Concretely, this is achieved by configuring traffic of higher ACs with shorter arbitration inter- frame space (AIFS).

2.4 Video Streaming Technologies

There are different video streaming technologies and techniques. The audio stream can be compressed to make the file smaller using an audio coding format such as MP3 or other codification that compress the audio.

The video stream is compressed using video coding format such as H.264, HEVC, VP9 or VP8 and then are assembled in a bit-stream with MP4, FLV, WebM, ASF or ISMA.

For video stream protocols there are Real-Time Messaging Protocol (RTMP) (Adobe), Real-Time Transport Protocol (RTP), HTTP Live Streaming (HLS) (Apple) and non-proprietary formats such as MPEG-DASH that is responsible to enable adaptive bit-rate streaming over HTTP as an alternative of using proprietary protocols. This work will use RTMP protocol for video stream.

2.4.1 H.264

H.264-Advanced Video Coding (AVC) was defined under a Joint Video Team (JVT), and comprises two layers: the Video Coding Layer (VCL) and the Network Abstraction Layer (NAL). The former is used to create a coded representation of the source content, providing flexibility and adaptability to video transmission, while the latter is used to format the VCL data and to provide header information on how to use the data for network video delivery. Each image is partitioned into smaller coding units (macroblocks) in VCL, which are themselves comprised of independently parsable slices. These slices are further partitioned into three groups for flexible partitioning of a picture.

NAL units are the video data encoded by VCL plus an one-byte header that shows the type of data contained in the NAL unit. One or more NAL units can be encapsulated in each transport packet. NAL units are classified as VCL NAL units (coded slices or coded slice partitions) or non-VCL NAL units (containing information such as sets of parameters and Supplemental Enhancement Information (SEI)). Each coded video sequence is an independently decodable part of a NAL unit bit stream, and starts with an Instantaneous Decoding Refresh (IDR) access unit. The IDR access unit and subsequent access units are decodable without decoding any previous pictures of the bit stream. The NAL payload is transmitted with different priority.

2.4.2 HEVC / H.265

The High Efficiency Video Coding (HEVC) standard is intended at decreasing the bandwidth requirements of video services by providing a 50% increase in compression efficiency over previous H.264-AVC standard, while keeping the same level of perceptual visual quality. Similarly to H.264-AVC, HEVC consists of a VCL and a NAL layers.

The coding structure used in a VCL layer for each picture is significantly changed. While in H.264/AVC each picture is divided into macroblocks (with 16x16 luma samples), which can be further divided into smaller blocks (16x8, 8x16, 8x8, 8x4, 4x8 and 4x4), in HEVC pictures are split into Coding Unit (CU) treeblocks of up to 64x64 luma samples, with the highest level of the treeblock structure referred to as the Largest Coding Unit (LCU). Tree block structures can be recursively split into smaller CUs through a quad-tree segmentation structure – CUs can vary from squared 8x8 to 64x64 luma samples. The higher compression gains can be achieved using larger CUs on homogeneity regions within a picture with little or no motion between two adjacent pictures, when using intra-prediction and transforms.

2.4.3 Real-time Transport Protocol

The Real-Time Transport Protocol is a transport protocol for delivering audio and video over IP networks. It specifies a structure to send audio and video and typically runs over User Datagram Protocol (UDP). RTP is used with the RTP Control Protocol (RTCP) . RTP is responsible to carry the media streams (video and audio) and RTCP is used to monitor transmission statistics and QoS that empowers the ability of synchronization of multiple streams.

2.4.4 Real-Time Streaming Protocol

The Real Time Streaming Protocol (RTSP)[24] is an application-level protocol for control over the delivery of the data with real-time properties.

RTSP provides an extensible framework to enable controlled, on-demand delivery of real-time data, such as audio and video. Sources of data can include both live data feeds and stored clipes.

This protocol is intended to control multiple data delivery sessions, provide a means for choosing delivery channels such as UDP, multicast UDP and Transmission Control Protocol (TCP), and for choosing delivery mechanisms based upon RTP.

It allows to create a side channel to make user-control operations: play, stop and pause. It allows an out-of-band control and the content descriptor is downloaded (which tells what is the content, what is the port for the session and other settings).

2.4.5 Real-Time Messaging Protocol

RTMP is a protocol developed by Macromedia (nowadays Adobe Systems[1]) for streaming audio, video and data to the Internet focused initially on Flash Player. It is very simple and transports over TCP on port 1935.

The protocol is now open to the community since 2009[25].

2.5 Work topics

This dissertation address intends to work in mechanisms of QoS for uplink in a service provider SDN network. In fact, uplink of live video content needs more research and work to be done. When a network it is full of, it is very important to take care of low latency flows (like video), for example, with QoS.

[1] `http://www.adobe.com/pt/`

CHAPTER 3

Solution

The purpose of this chapter is to present the solution for the motivation described before. First, the concept architecture, the Video SDN Application (VSA) module, how this work has been included in the VDSNet project.

The VDSNet project has been introduced before, the main goal for the project is to create mechanisms with QoS for video streaming scenarios where mobile users are uploading content for the services through the network operator.

The developed solution and the VSA module has been included as the concept architecture for the VDSNet project. The implementation will be presented in the following chapter.

3.1 Solution Architecture

The work for this dissertation joins with the problems and solution developed for the VDSNet Proof of Concept and architecture. Therefore, the architecture focuses on video streaming scenarios where mobile users are the video senders and SDN technologies are used to improve the quality of service and user experience.

In particular when live video streaming services with low encoding/ transcoding latency are important, the improvement of the uplink operation will translate into network efficiency and improved overall user experience as the improvement can be translated into prioritization of the traffic.

The upload scenario mentioned above has not yet been fully addressed in the research community, at the current date, and the scenarios and issues were not addressed yet too. This work will try to explore and extend the benefits of SDN in the upload with a mobile video device.

3.1.1 Considered services

The work followed a service-oriented approach considering the open APIs and associated control provided by SDN for end-to-end QoS in video streaming technologies. It considered

RTMP as the standard for the final demo of the work and tests, but other protocols and content should fit in the work as well, since the prioritization is based on the flow tuple[1].

3.1.2 Architecture composition

This work follows the architecture depicted in Figure 3.1. It comprises three modules residing in the network: VSA, SDN Controller and the Monitor application. For the VDSNet project, this work developed and contributed for the architecture, Video SDN Application module, interaction with the SDN Controller and SDN based network deployment.

The main tasks of each module can be summarized as:

- **VSA** is the responsible central node of the network policing, control and orchestrating. It will talk with the other two network elements and make network considerations and decisions using QoS.
- **SDN Controller** controls and collects statistics about the flows in the switches at the Core and Access network.
- **Monitor application** at the Radio Access nodes, the APP is responsible to monitor video flows and send the information about them to the VSA.

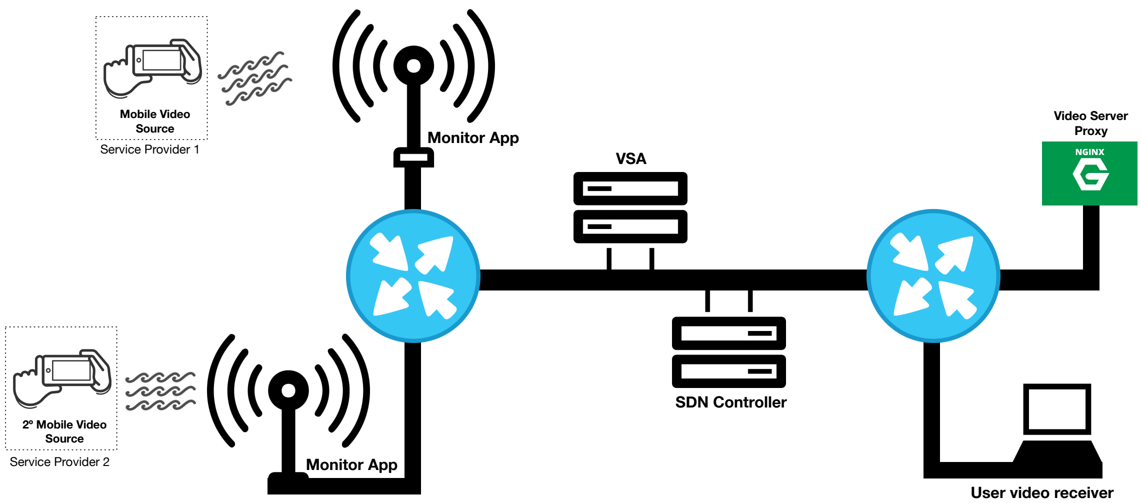

Figure 3.1: VDSNet project architecture

3.1.3 Modules description

a) SDN Controller

The SDN Controller is the centralized view of the SDN network, responsible for being able to collect network statistics from the network switches and terminals and to control prioritization of the services according to the policies received.

So, in this work, it was considered two inner-functions for the controller:

- **Prioritization** capabilities, determine each flow must go first to output node using the receiving orders.

[1](IP, TCP Port)

- **Flow management** stores the flow definitions sent by other entities and applications through the northbound interface, enabling the SDN Controller to do dynamic flow control.

b) Video SDN Application (VSA)

VSA handles requests for changing the QoS of mobile-originated live stream sent by service providers or other elements in the network that have the ability to talk with the VSA.

The requests can be made from authenticated mobile Content Service Providers, network elements or other Broadcasters depending of the final use case.

With that, VSA can make prioritization of certain types of flows account the content-specific policies.

During the processing of every request, VSA can deliver to the SDN Controller actions (expressed in terms of QoS) associated with the requirements needed by the request entity. Then the SDN Controller is responsible to spread the actions through all the network elements.

VSA also contains interfaces in order to receive that requests and to provide information about the flows in the database. Figure 3.2 shows how the requests between all the elements for better quality of service are processed and the order that is expected.

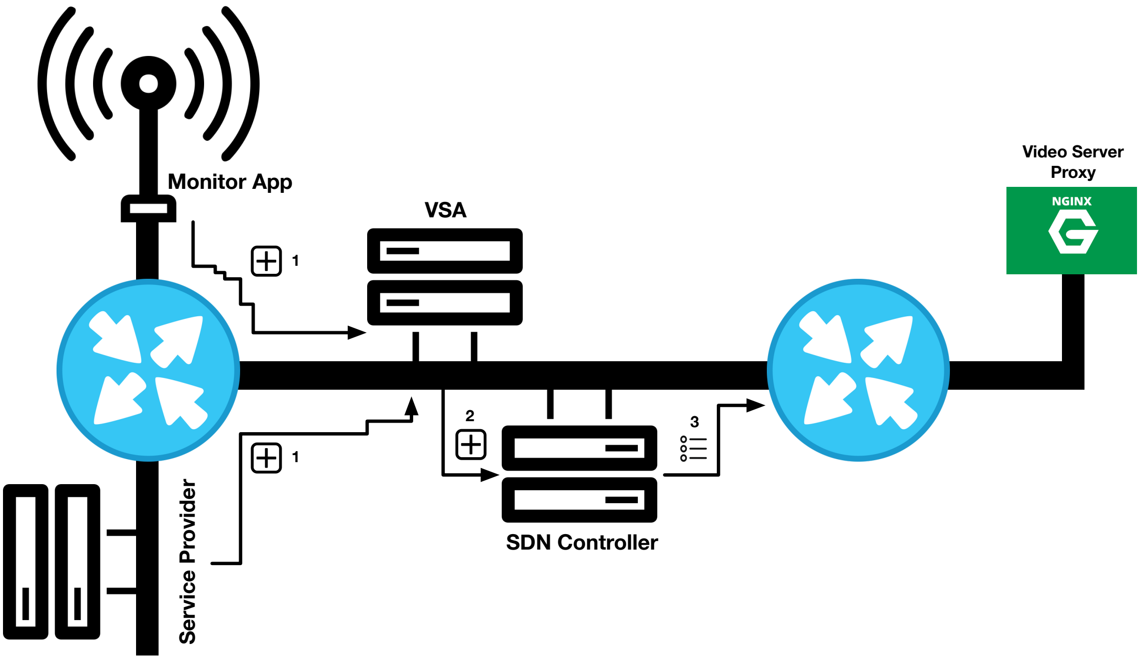

Figure 3.2: VSA requests interaction

The VSA can be extended depending of the final use cases. It can have different types of queues, quality-of-service (depending how much the service provider pays for the treatment or other SLA).

c) Monitor application

The monitor application of the VDSNet project intends to track the mobile video flows and to send requests to the VSA for better treatment.

3.1.4 Use cases

a) Use case description

John has just acquired his new device, which features connectivity to multiple radios (e.g. 4G, IEEE 802.11ac, Bluetooth) and Full HD (1080p) video recording capabilities. He has promised his family and friends he would do a live transmission showing them the beautiful views in his current trip. For this, he will take advantage of the device's embedded live video transmission capability, leveraging the highly adaptive capabilities of it's network's operator – recently upgraded with this work architecture - to deliver his video with superior quality, in a seamless experience.

In a similar scenario, Anna, another mobile user, will also use the network for transmitting her video in the same geographical area as John. The main difference is that Anna's service provider has no access to this work APIs; thus, while John's video will be delivered using SDN-based priority approach, which integrates quality of service for the video flow, Anna's video will be delivered using a legacy solution – for instance, exclusively relying on the application robustness and intelligence, typical from adaptive streaming solutions). After a pre-determined time, congestion will be introduced at the mobile sender's access network, which will affect the two users differently.

b) Use case goals

This scenario intends to demonstrate how it is possible to optimize the user experience using SDN in mobile wireless architectures and by allowing network operators and service providers to unify the control plane.

Moreover, it intends to show different benefits of the solution for the provisioning of services which depend on quality assurance of uplink transmission, such as automation and customization of service differentiation.

Finally, the selected scenario intends to show the advantages that SDN brings to the accomplishment of dynamic SLAs for different service providers.

3.1.5 Stakeholders

The architecture definition for the project and this work had a significant impact from the study of the business ecosystem and definition stakeholders.

With the identification of multiple business positioning and relationship options, which are - and will be - shuffled with SDN/Network Function Virtualization (NFV) technologies [8], we can identify three different stakeholders at least:

a) Network operators

With the grow of over-the-top services network operators have reflected in revenue loss[26] due to the increasing need of bandwidth by that services and the necessity of maintain the quality standards.

It means that Over-the-top (OTT) services are using network resources to deliver content and, with the growth of that services, the costs of maintaining a sustainable network are higher due to higher bandwidth requirements (increasing video OTT services) and the OTT services do not pay to use the operator network.

With that, the network operators can give Quality of Service for the OTT services if they pay for that.

b) Service providers

Service providers (like Facebook, Instagram and others) sometimes have difficulties in delivering high quality content with the better QoS and experience for the end users.

Services fight in the same level to delivery the content, side by side, with other services and they only have control of the channel between end users and service, not at the network. With this work, service providers can request better treatment by the network provider for the service channel.

c) Mobile video streaming users

When too many people are in the same area, for example in musical festivals, the mobile network tends to be very poor. The network gets very bad if many of these attendees start a video stream session of the artists show.

With this work, network operators can benefit from better reputation if they use quality of service in favor of the most used service in the network. For example, 70% of the attendees use service A, so the network operators can provide better QoS for service A and the users from other network operators will think that the others have excellent network.

In the other hand, in the same scenario, service providers can pay to have better QoS and that will translate in better relationship with the user and they will think that only that service works very well in that circumstances.

CHAPTER 4

Implementation

4.1 Adopted Technologies and Tools

Many technologies were used to demonstrate the proof of concept. The implementation adopted for the work it is not the only that can be adopted, but fits very well for what this work intends to show.

4.1.1 Mininet

Mininet was previously introduced in section 2.2.4. It was used to create a Layer 2 (L2) network, with switches and hosts. The network created by Mininet by default is a simple one, with a basic topology.

Since the proof of concept intends to interact with the user mobile phones, it was very important to connect the Mininet network with an physical network. With that, it is easy to transmit video streams from the mobile phone (physical network) to a Mininet network (SDN network).

Initially, it was not easy, but bridging an interface from the computer that runs Mininet to one network switch allowed to make communications between the SDN network and the physical network allowed this to be possible. The Mininet hosts will be able to communicate with the mobile phones hosted by WiFi (and vice-versa).

As shown in Figure 4.1, the initial setup using Mininet is very simple and the hard task is to make a bridge between the Mininet switch and the laptop interface. Mininet has a brief of how to do that in its website [1].

After the connectivity between the physical network and Mininet network is successful, different programs with specific IP and ports are required to run in every mininet host.

[1] `https://github.com/mininet/mininet/blob/master/examples/hwintf.py`

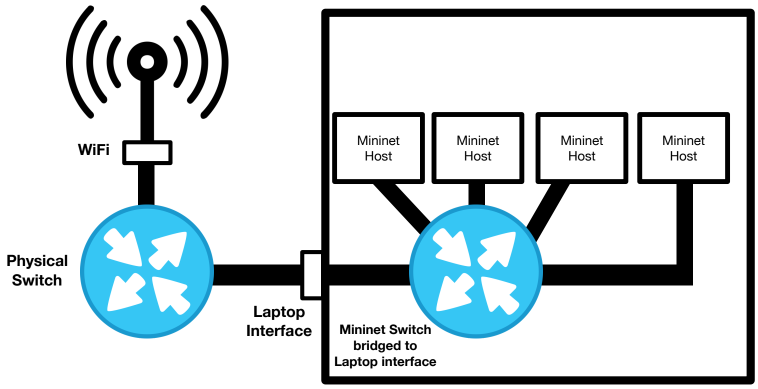

Figure 4.1: Mininet initial setup

a) Mininet hosts

Mininet allows that each host runs a program from the operating system but it is not possible to jail a program[2] to a specific host as an independent instance.

That limitation leads to a problem, it is very important that each Mininet host is an independent instance, in order to bind the instance to a specific interface and port, enabling to act like a normal virtual machine.

b) Switching

By default, Mininet uses Open vSwitch Kernel Switch[3] for switching.

An other type of switch is the user-space switch[4], that runs in user space instead of kernel space. By running in user space[5] it means that it will have a higher delay, the packets will be more variable and by being in user-space the process can be scheduled to execute later because of concurrency paradigms with other user-space processes.

Running Mininet switch in user space:

```
# sudo mn -switch user -test iperf
```

Dpctl[6] is a management utility that enables some control over the OpenFlow switch. Mininet uses that utility to control the standard Mininet Switch (Open vSwitch Kernel Switch).

With that utility it is possible to add flows to the flow table, query for switch features and status, and change some other configurations.

Example:

```
# dpctl unix:/var/run/s1.sock stats-flow
```

Output result: Appendix 7.1.1

[2] http://mininet.org/walkthrough/#run-a-simple-web-server-and-client
[3] https://github.com/mininet/mininet/blob/master/mininet/net.py#L116
[4] http://mininet.org/walkthrough/#interact-with-hosts-and-switches
[5] https://www.kernel.org/doc/Documentation/networking/openvswitch.txt
[6] https://github.com/CPqD/ofsoftswitch13/wiki/Dpctl-Documentation

c) Simple topology

A simple Mininet topology is started with four hosts, one switch and with a remote controller, by doing:

```
# sudo mn -topo=single,4 -mac -controller=remote
```

It will have an architecture like in Figure 4.2. As described, the SDN Controller is listening at TCP Port 6633, the Open vSwitch switch is listening at TCP Port 6634 (dpctl makes use of that port to make control of the switch). Each host is connected to the Open vSwitch switch and for each host one interface in the main host will be created.

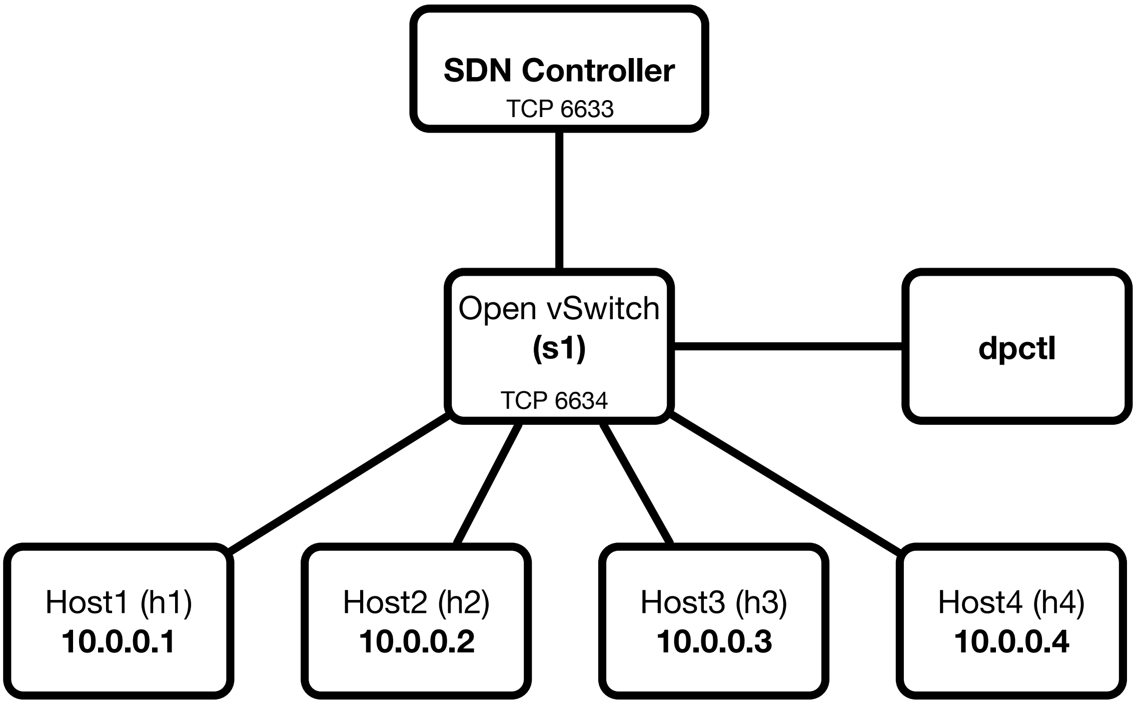

Figure 4.2: Mininet architecture

4.1.2 Open vSwitch

Mininet uses Open vSwitch, which was already introduced in the Chapter 2.2.3. Mininet switches make use of Open vSwitch for switching purposes, so it is important to understand how OVS can be useful for the work.

a) Command line tools

The ovs-ofctl[7] program is a command line tool for monitoring and administering OpenFlow switches. It is capable of showing the current state of an OpenFlow switch (configuration and table entires) and it should work with any OpenFlow switch.

There are some ways of connect the tool to the switch:

- ssl:ip[:port] should be provided with a –private–key, –certificate and –ca–cert if needed.
- tcp:ip[:port] ip = ip address (IPv4 or IPv6).

[7] `http://openvswitch.org/support/dist-docs/ovs-ofctl.8.pdf`

- unix:file, a unix domain server socket named file
- file shortcut for unix:file

Other important tool is ovs-vsctl[8] which provides a high-level interface to configure the ovs-vswitchd database. It connects to the ovsdb-server process that mantains the Open vSwitch configuration database and applies changes or queries to the database.

Thus there are some important commands for the work, such as:

- **ovs-ofctl add-flow** add flow to the switch
- **ovs-ofctl del-flows** delete specific flow from the switch
- **ovs-ofctl dump-flows** delete all the flows from the switch
- **ovs-vsctl set port** allow to modify the port, for the work it will be used to create a linux-htb[9] QoSs record that points to queues.
- **ovs-vsctl clear Port** allow to clear the queues configured within the work
- **ovs-vsctl list qos** allow to list all the QoS records applied
- **ovs-vsctl list queue** lists all the queues created

b) Quality of Service (QoS)

Let is assume that there is a setup as described in Figure 4.3. There are two Mininet hosts connected to an Open vSwitch switch and a network measurement tool (iPerf)[10].

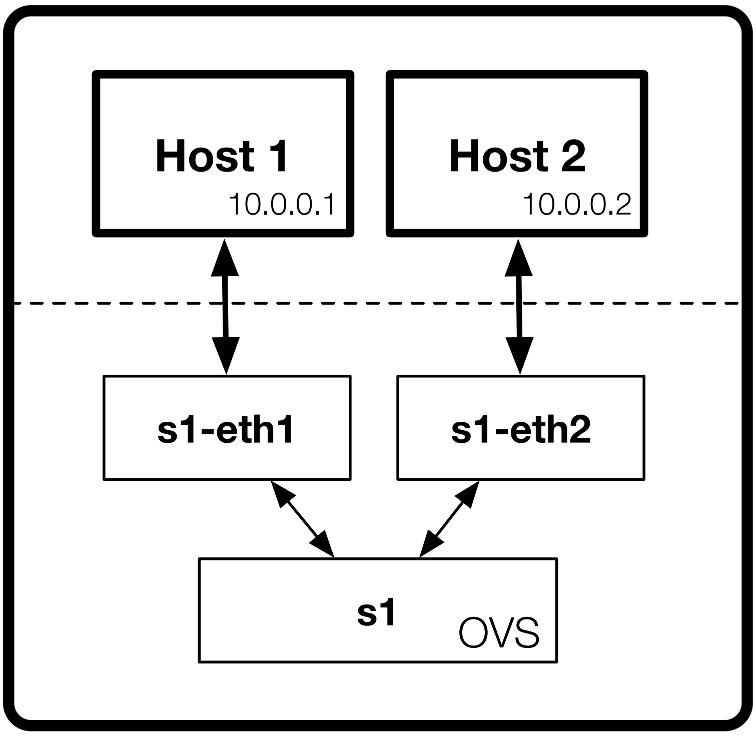

Figure 4.3: Quality of Service using Open vSwitch

[8] `http://openvswitch.org/support/dist-docs/ovs-vsctl.8.txt`
[9] `http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm`
[10] `http://docs.openvswitch.org/en/latest/howto/qos/`

In order to implement Quality of Service in an Open vSwitch (OVS) Switch, it is necessary to create different queues and policies. One queue will have limited bandwidth and the other one will have no bandwidth limitation.

OpenFlow version 1.3 provides limited support for QoS using queues and meters. Queues are created and configured outside the OpenFlow protocol and then used in queuing the flows.

Using the ovs-vsctl tool it is possible to verify if the QoS rules and queues lists are empty..

Listing QoS:

```
# sudo ovs-vsctl list qos
```

Listing Queues:

```
# sudo ovs-vsctl list queue
```

Listing the flows in the OVS swicth by priority:

```
# ovs-ofctl dump-flows s1 -rsort=priority
```

After that, using the command line it should be created two queues in the interface s1-eth1 and in s1-eth2 with the policies described above.

```
# ovs-vsctl set port s1-eth1 qos=@newqos - -id=@newqos create qos type=linux-htb
other-config:max-rate=100000000 queues=1=@q1,2=@q2 - -id=@q1 create queue other-config:min-rate=10000000
other-config:max-rate=10000000 - -id=@q2 create queue other-config:min-rate=100000000
other-config:max-rate=100000000
```

```
# ovs-vsctl set port s1-eth2 qos=@newqos - -id=@newqos create qos type=linux-htb
other-config:max-rate=100000000 queues=1=@q1,2=@q2 - -id=@q1 create queue other-config:min-rate=10000000
other-config:max-rate=10000000 - -id=@q2 create queue other-config:min-rate=100000000
other-config:max-rate=100000000
```

After that the flows will be enqueue when they arrive at the s1 switch. Creating a queue does not mean that the flows were limited, they must be sent through the queue in order to be limited.

One of the two hosts will have unlimited bandwidth and the other one will be limited at 10Mbps, so all the flows with destination to the Host 2 will be queued. Using ovs-ofctl the flow will be added to the flow table and then queued in queue number 1, the other 2 is to specify the output port. Priority will be set to 50 (although it is a low priority number the match statement is very specific so it will match first than flows with wildcards).

```
# ovs-ofctl add-flow s1 priority=50,tcp,nw_dst=10.0.0.2/32,actions=enqueue:2:1
```

We can validate the approach by running the following command:

```
# iperf
```

And the result should be:

```
*** Iperf: testing TCP bandwidth between h1 and h2
.*** Results: ['9.42 Mbits/sec', '12.6 Mbits/sec']
```

As can be concluded, the link between the h1 and h2 is limited at 10Mbits/sec.

To delete the flows in the OVS switch:

```
# ovs-ofctl del-flows s1
```

4.1.3 OpenDayLight

This work required to use a SDN Controller, and OpenDayLight was selected for the development of the proof of concept, described in the Chapter 2.2.1.

In Figure 4.2, the specified a SDN Controller that listens at TCP Port 6633, can be the OpenDayLight. All the Open vSwitch switch will query the OpenDayLight controller.

Using the installation guide[11], the OpenDayLight has been installed using the commands described in the Appendix 7.1.2.

4.1.4 Containernet

Containernet has been previously mentioned in chapter 2.2.5 as a project based on Mininet that intends to support Docker as Mininet hosts in a Mininet network.

a) Architecture

Let is assume a basic example as shown in the Figure 4.4. Two docker containers will be connected within the Docker network. With dockernet, each docker container has a new interface that connects to the OVS switch and has a previously assigned IP address. That feature enables to connect the Docker container to the switch S1 (OVS).

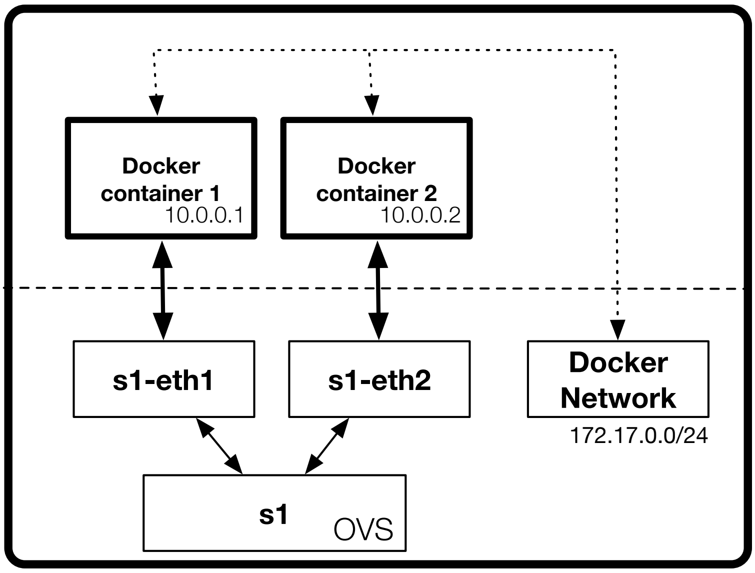

Figure 4.4: Containernet connect example

b) Containernet hosts

A docker container provides an isolated environment to run our applications. The container will act as an independent process and like a normal virtual machine, for example, biding to the instance specific ports.

That isolation benefits who wants to simulate independent applications and environments in a SDN Network, like the work developed in this dissertation.

[11] `https://www.opendaylight.org/sites/opendaylight/files/bk-install-guide-20150831.pdf`

4.1.5 Docker

The OpenDayLight Controller instantiation takes too long in a nowadays normal environment, in order to simplify that, it was made a docker image (described in Appendix 7.1.2) that will provide a fast and clean boot for the Controller.

Other images were also developed, which are described in the following sections.

4.2 Proof of Concept Phase 1

In the previous section relevant technologies for the proof of concept were introduced. The implementation of this first proof of concept intend to be able of demonstrating the pros, cons of the architecture described and get important results to be further analyzed.

Using the previous introduced technologies, the first concern was how to connect a physical network with a SDN network and that possibility was described in section 4.1.1. It is possible to bridge an OVS switch to a host interface, and that solves the problem of connecting the physical network (Wi-Fi) and the SDN-based network.

Besides that, the second implementation concern was solved using the Containernet described in section 4.1.4, the Mininet hosts can be isolated from each other and act independently.

This proof of concept was been presented in 15th May 2017 at Lisbon Altran Portugal offices and for four schools at ATNoG (Intituto de Telecomunicações - Polo Aveiro), visiting the laboratory (it was very interesting to verify the teenagers audience reaction to the video stream with and without our solution).

4.2.1 Architecture

The architecture depicted in Figure 4.5 shows how the first phase PoC was made. In the beginning of the work the architecture requirement was that the mobile video source must be over radio (Wi-Fi or 3G/4G) and for that, it was considered that the video source was a station using Wi-Fi.

For better understanding of the results it was considered that the traffic generator is in the same level of the mobile video source, but not over Wi-Fi. It is not relevant because the quality of service is only applied at the OVS switch instance and the limitation over Wi-Fi is out of the scope of this work. The user video receiver is watching the video stream using a ordinary stream player, for example, VLC[12].

In order to provide Wi-Fi and LAN connectivity for the desktop clients, a normal Access Point was used that has LAN ports. The physical host where the SDN network will be instantiated is connected to the AP using one of the LAN ports available.

As described above, the physical host is connected to the Wi-FI Access Point via one of the available LAN ports, and the OVS Switch is in bridge mode of that interface, connecting the SDN network with the Physical network. With that, by using Containernet, the modules images (VSA and Monitor application) and the OpenDayLight docker image were created.

[12] `http://www.videolan.org/`

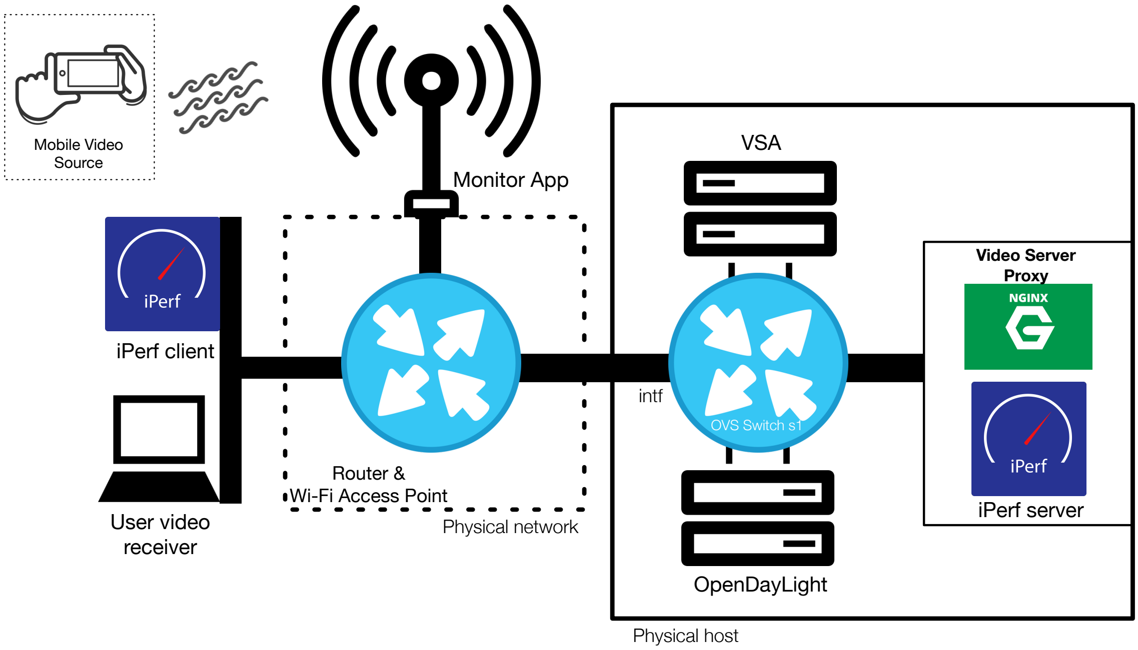

Figure 4.5: Architecture Phase 1

Those images provide a fast startup of the system and, for OpenDayLight, provides a easy startup without failures.

Last but not the least, the video server proxy (stream rendezvous point) and the iPerf server (traffic generator receiver) has a docker image too.

4.2.2 Components installation

The PoC components were installed in a Ubuntu[13] operating system running a VirtualBox[14] VM. For easier development, it was been downloaded the Mininet VM[15], then made some changes described in the Appendix 7.

This PoC, was in addiction more components like, an OpenDayLight, iPerf and RTMP-Server docker image, that will be described in the next section.

4.2.3 Docker images

Docker has been introduced in the chapter 2.1.1. This section intends to describe how the OpenDayLight , RTMP-Server and iPerf docker image were implemented, and what files and contents were important to make the images.

a) OpenDayLight docker image

For the OpenDayLight docker image, it is important to highlight that the Dockerfile, present in the Appendix 7.1.2, expose eleven ports and the port 6633 it is used by the OVS switch to query the forwarding table. Exposing that ports will make them accessible for the others Dockernet hosts.

[13]`https://www.ubuntu.com/`
[14]`https://www.virtualbox.org/`
[15]Mininet 2.2.1 on Ubuntu 14.04 LTS

Placing the file *"org.apache.karaf.features.cfg"* at the folder /odl/etc will make that the require features will automatically be installed. For more details please visit the chapter 4.1.3.

To build the docker image run inside the work folder:

```
# docker build -t vdsnet/odl .
```

b) RTMP Nginx server with iPerf

Then a docker image with RTMP Nginx server and with iPerf was made, as described in the Appendix 7.1.3.

For this image it is important to highlight too that there are four different ports exposed. The 5201 is used by iPerf and the others two by RTMP NGINX. The nginx configuration is copied for the container as *"nginx.conf"* file, and the configuration is explained bellow.

The important part of this configuration is the RTMP entry, configures a RTMP rendezvous point that can record what is reproduced and send to the user that request to receive the live-stream.

For upload to the NGINX RTMP Proxy server, the user must specify the url like "rtmp://nginx_ip/live/stream" and to receive it will be "rtmp://nginx_ip/live/stream" both points only need to know the server IP.

4.2.4 Containernet

Containernet has been previously introduced in chapter 2.2.5. It is a fork from the Mininet project so, using a Mininet Virtual Machine, the Containernet can be installed using the commands listed in the Appendix 7.1.3.

4.2.5 Video SDN Application

The module VSA has been previously introduced in chapter 3.1.3. For this first proof of concept, a command line tool, web service (Figure 4.6) and a dashboard (Figure 4.7) have been developed.

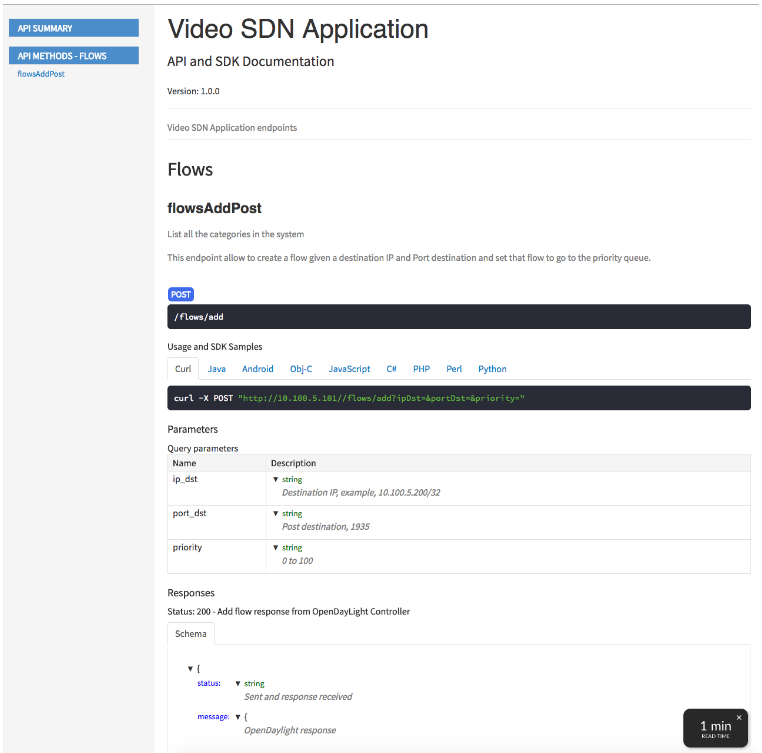

Figure 4.6: VSA API

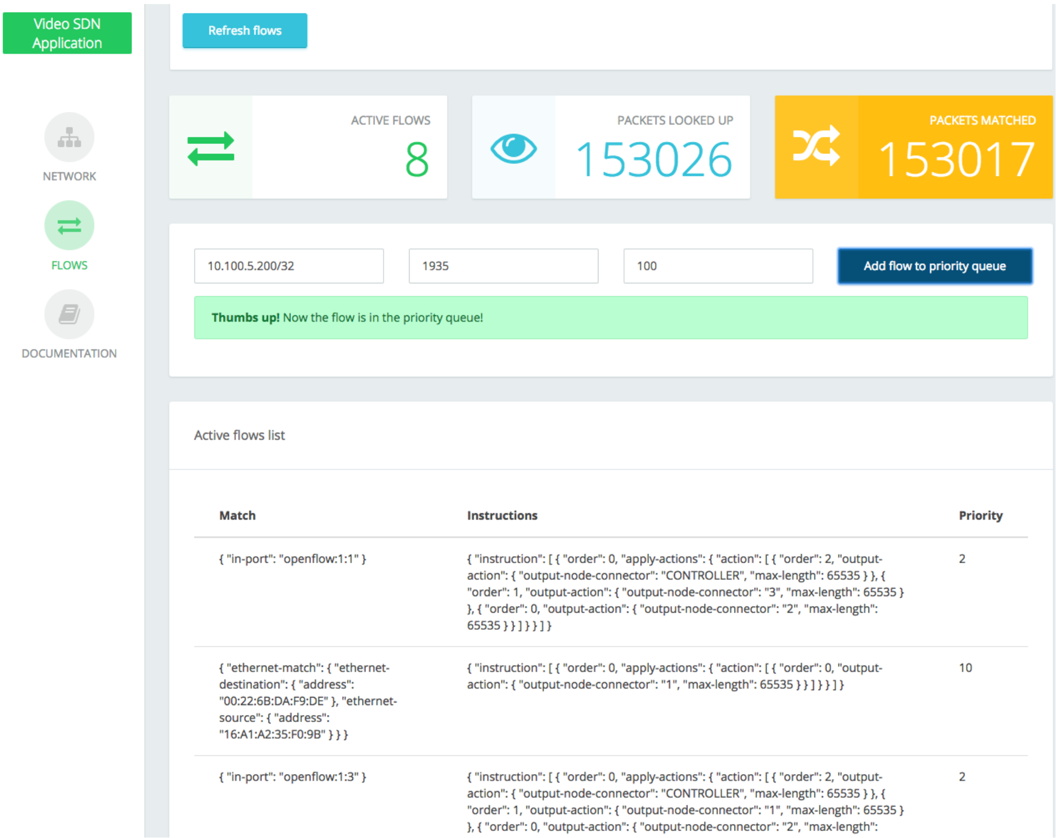

Figure 4.7: VSA dashboard

a) Dashboard and Web services

The implemented dashboard has been made using jQuery, HTML and some bootstrap CSS.

Some requests are made directly to the controller, such as the statistics numbers.

The API has been made using Flask[16] (Python web framework) and there are some interfaces:

- **/flows/add** this interface provides to other work modules a way of prioritize a flow
- **/flows/docs** this endpoint provides a documentation about how to use the previous interface

b) Command line interface

The command line arguments for the *vsa.py* are:

- **python vsa.py –web** starts the dashboard and API
- **python vsa.py –list_routes**
- **python vsa.py –ip_dst=IP/32 –port_dst=IP/32 –priority=0 to 65535**

4.2.6 Interaction diagram

The proof of concept developed has an interaction diagram that proves how the work architecture can provide better quality of service for the video stream mobile users. It is not the only way the work architecture can be implemented, in the next chapters it will be detailed how this architecture can be flexibly changed according to the scenario and use cases intend.

In this proof of concept, the VSA only works as a forwarder and as a system provisioner but there are other ways of operation and more decision intelligence can be introduced like verify the requests complain the agreements between network operator and the services.

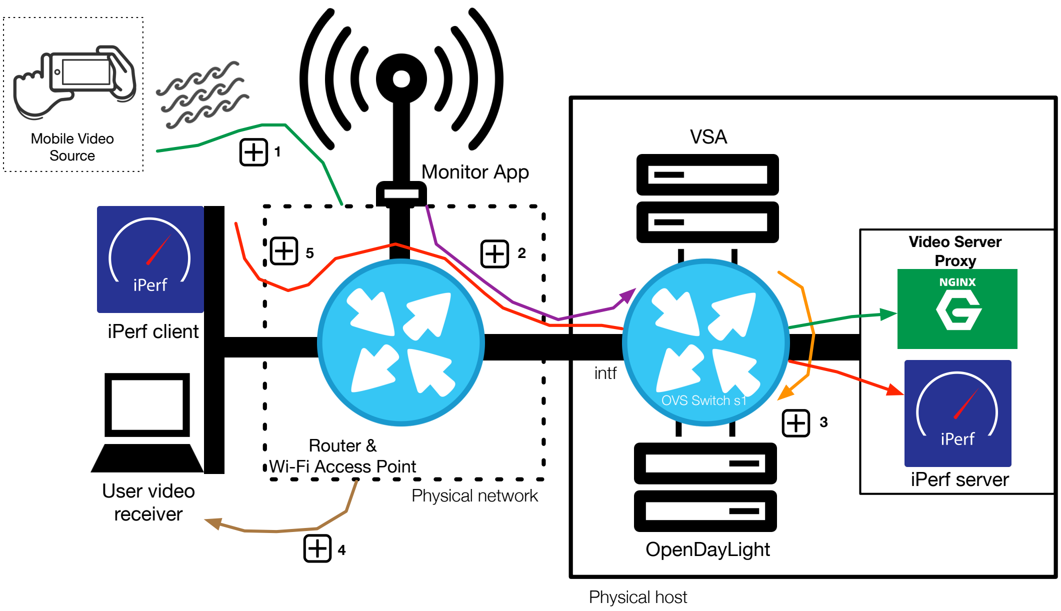

Figure 4.8: First phase architecture interaction

16 `http://flask.pocoo.org/`

Figure 4.8 shows the interaction described in the following steps:

1. Mobile video source streams to the Video server proxy via the Wi-Fi network;
2. Monitor module application detects video flow and sends a notification to the VSA in order to prioritize that flow;
3. VSA sends an order to the OpenDayLight controller;
4. The user video receiver starts to receive the stream video flow from the video server proxy after a request;
5. The iPerf client (bandwidth measurement tool) starts a bandwidth measurement session with the iPerf server;

The QoS implemented is based on queues and bandwidth limit described in the chapter 4.1.2.

The behavior expected for the packet losts for this proof of concept and the first flow sent by the mobile video source, it will have dropped packets because the traffic generator begins while the flow is sending, and the queue in the output node to the host that has the Video Server Proxy and iPerf Server is limited at 10Mbps.

After that, the video stream will go to the best QoS queue and no packets will be lost.

4.3 Proof of Concept Phase 2

After the first phase of the PoC, it was made a second implementation but this time with a Raspberry Pi instead of a physical x86 host, thus allowing the evaluation of the concept using hardware with fewer resources.

An other requirement for the second phase is that the PoC can be remotely controlled via a mobile application. The Wi-Fi Access Point is provided by the Raspberry Pi. In this phase the monitor application module was removed.

4.3.1 Architecture

For the second Proof of Concept, it was initially thought to divide the modules of the first PoC into many Raspberry Pis.

Nonetheless, after some research, it was verified that Docker now supports Advanced RISC Machine (ARM) Raspberry architecture, so it was decided to use the same type of architecture done so far. Using the same components, it will be possible to understand the differences in a all-in-one architecture and a x86 computer connected to an AP.

The architecture depicted in Figure 4.9 shows how the second phase PoC was made. As described in phase one, in the beginning, the architecture requirement was that the mobile video source must be over radio (Wi-Fi or 3G/4G) and for that, it was considered that the video source was a station using Wi-Fi.

In the first phase, an Access Point that provides Wi-Fi and LAN connectivity for the desktop clients has been used, but in phase 2 it was used only Wi-Fi and that Wi-Fi has been provided by the Raspberry Pi 3 as Access Point.

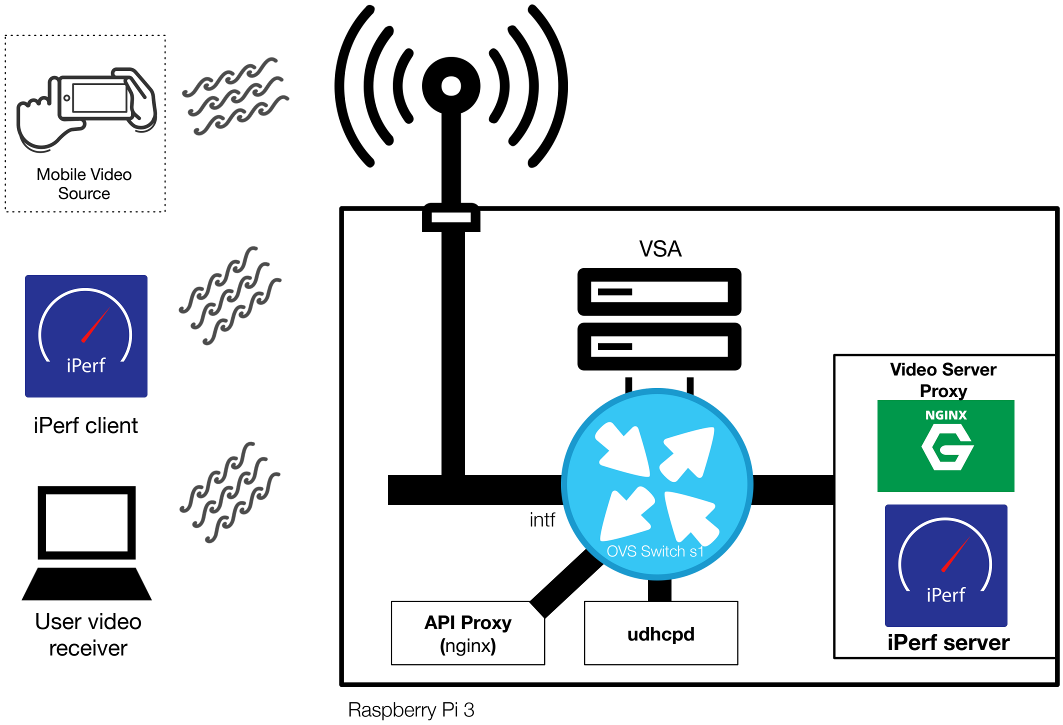

Figure 4.9: Second phase architecture

So, the bridged interface now is the Raspberry Pi Wi-Fi interface, connecting the SDN network with the Physical network. Containernet network implementation stays the same, and the docker images are equal too.

4.3.2 Implementation differences

There are some differences in the architecture that led to some implementation assumptions being changed. First, the Wi-Fi interface of the Raspbery Pi will be used.

Hostapd[17] receives the association requests to the Wi-Fi network, handles them and the client is connected without the need of an IP address in that interface. The OVS switch is bridging the Wi-Fi interface, thus it will not have an IP address.

After an association, the client will need a Dynamic Host Configuration Protocol (DHCP) offer, since the connected client sends a DHCP request in order to have an IP address. With that requirement, another change in architecture is the introduction of a udhcp[18] docker image for ARM.

a) Hostapd

The steps that describes how the hostapd has been configured using a Raspberry Pi 3 can be consulted in the Appendix 7.1.4.

b) Containernet

Containernet, at the time of this work, did not support Raspberry by default. The development of this PoC contributes with the open source project in GitHub[19], and there are some modifications that have to be done in order to work with Raspberry Pi 3. That modifications and work can be consulted in the Appendix 7.1.4.

c) Docker images

The iPerf and RTMP Server docker images stayed the same but there are new docker images built: dhcp and api-proxy and the implementation can be consulted in the Appendix 7.1.4.

4.3.3 Startup script

One of the requirements of this phase is that the network starts automatically and can be remotely managed since the PoC is all-in-one solution.

For that requirement it was created a startup python program that initiates the Containernet network, execute the necessary verifications and makes the VSA ready to receive requests and manage the network.

Responsibilities of the startup script:

1. Clean the OVS Switch tables and queues configurations
2. Verify if the Wi-Fi is connected and the Containernet network can correctly operate
3. Creation of a tmux session with a window separated into two panes, one for the VSA network and other for other usage (example: iPerf client)
4. Start the VSA and SDN network

The implemented script can be consulted in the Appendix 7.1.4.

[17] `https://w1.fi/hostapd/`

[18] `https://udhcp.busybox.net/`

[19] `https://github.com/containernet/containernet`

4.3.4 Android control application

This second phase requires a remote control application, since the Raspberry will run the demo software and can only be accessed via SSH or other management program, is required to output the network information and create a channel to change the software state.

For that propose, it was used the framework PhoneGap[20] to create a multi-platform application running in an old android phone.

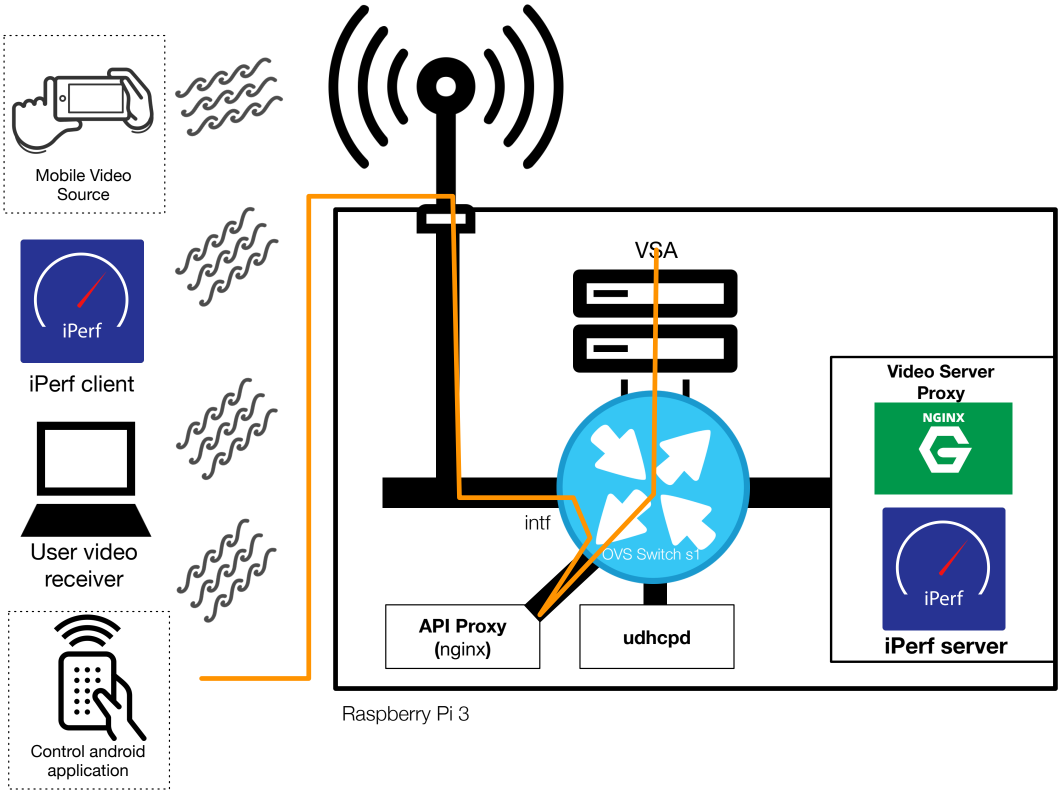

Figure 4.10: Control application connecting with VSA

Figure 4.10 depicts how the Mobile Control application can control the demo. The reason behind the control application connects with the VSA application is because the VSA already has methods that were relevant with the application usage, like adding a flow to the OVS Switch.

Responsibilities of the android application and new methods of VSA:

1. Network VDSNet restart
2. Network VDSNet stop
3. Get the report of all the components of the VDSNet network

4.3.5 Video SDN Application

Besides the responsibility the VSA already had, now it has more responsibilities that were described above, even through the VSA it is running at the Raspberry Pi host operating system and not inside the Docker container. Since the VSA needs to connect with the system

[20] `https://phonegap.com/`

software, it must be at the same level of the system python instance. And, for that, to make requests it must have an IP address inside the network.

Since the Raspberry Pi Wi-Fi interface does not have an IP address, it is required to create a docker container that proxies the requests to the localhost (raspberry IP address). Since the VSA runs with 0.0.0.0 (bind all interfaces), if the packet is forwarding for one of the reachable interfaces, sending it through a docker container it will have connectivity. That is why there is a new API Proxy built with NGINX.

4.3.6 Interaction diagram

For this second phase the interaction is very simple. Initially let is assume that all the Wi-Fi clients are connected and have an IP address.

The interaction diagram is depicted in the Figure 4.11 and will be described bellow:

1. The first interaction is by the Android Control application in order to get the output logs from the system PoC via the VSA.
2. After that, the mobile video source starts to transmit video to the Video server proxy
3. Then the iPerf client (packet generator), starts to measure the bandwidth available and many packets are lost by the mobile video stream
4. The control android application sends a request to the VSA to add the flow the priority queue
5. The VSA sends to the OVS the order to put the stream flow in the priority queue and then no packets are lost

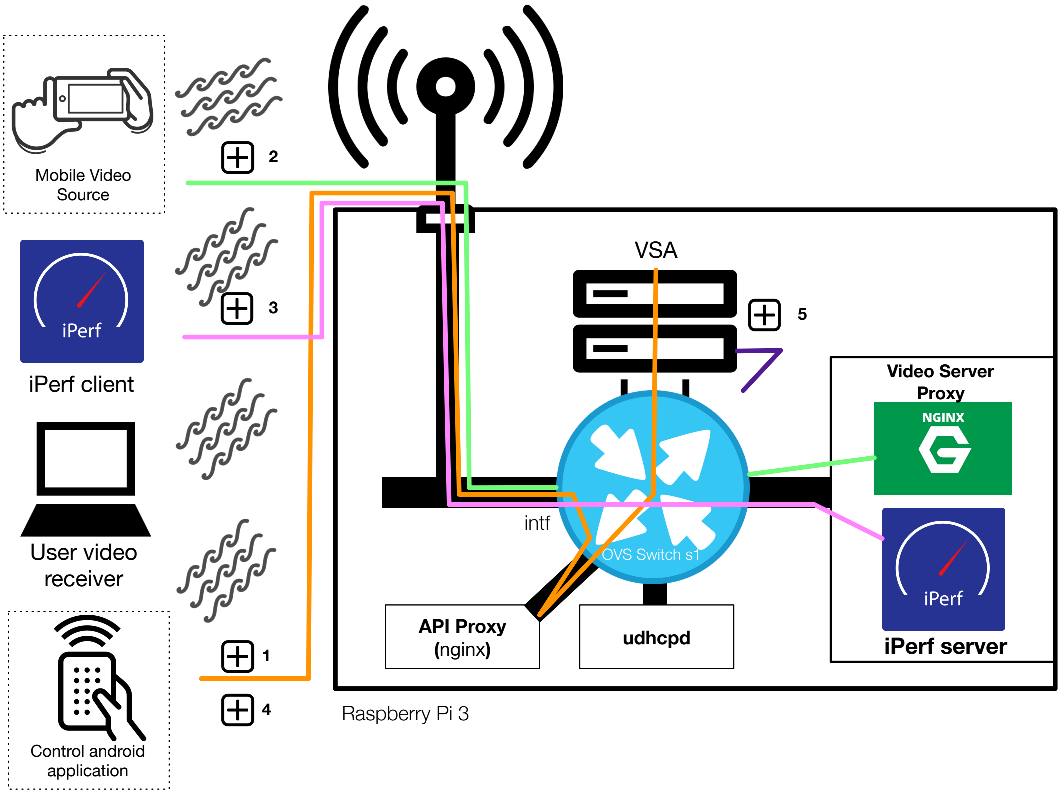

Figure 4.11: Control application connecting with VSA

CHAPTER 5

Evaluation and Results

In this chapter the two PoC developed and mentioned in the previous chapters will be evaluated then the results will be compared and some conclusions will be taken.

5.1 Deployment Scenario

For the evaluation of the solution developed, the two proof of concepts that were made will be compared.

In this section the network system will be evaluated (that hosts all the modules and the SDN Network) and the developed module VSA.

5.2 Performance Results

5.2.1 Evaluation details

For the two proof of concepts developed, the system and the VSA module will be evaluated over distinct metrics.

5.2.2 Evaluation metrics

Potential metrics that can be collected as indicators of the user satisfaction, network performance and respective improvement against legacy or sub-optimal alternatives that can be explored with this work are:

- PoC system startup time
- Number of packets lost with service prioritization
- Number of packets lost without service prioritization
- How long it takes to a service be prioritized

5.2.3 First phase results

The test bench makes use of the PoC scenario deployed for the first phase depicted in Figure 5.1. Some instrumentation changes were made in order to get the metrics requested.

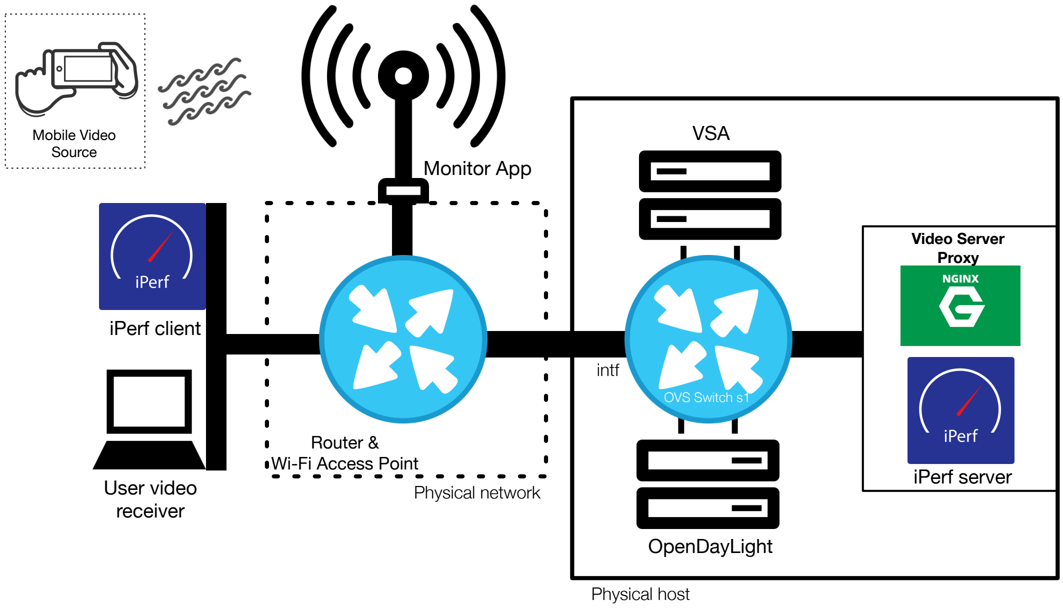

Figure 5.1: First phase PoC

a) Characteristics of the test bench stream

- **Streaming video resolution:** 852x532
- **text**
- **FPS:** 30/1
- **Bit-rate:** 2000 Kbps
- **Buffer size:** 2000
- **Codification:** Software (x264)
- **Streaming protocol:** RTMP
- **Standard network bandwidth:** 10Mbps
- **Streaming time:** 1 minute
- Streaming from the sender to the rendez-vous point
- Only one client watches one stream
- **Downscale filter:** Bicubic
- **Format:** NV12
- **Downscale filter:** Bicubic

b) PoC deploy virtual machine

The PoC scenario has been deployed in a Virtual Machine with the following configurations:

- **Main memory:** 8024 MB
- **Processors:** 2

c) Bench test order

- Instantiation of the system
- Instantiate N programs that uploads video to the service provider (N = number of client uploading streams)

- Start the traffic generator
- Start the uploading stream
- Count one minute
- Collect the results
- Run for each test bench 20 independent tests

d) PoC system startup results

The system startup time is independent of the clients that upload streams to the service provider. To get the following results in table 5.1, twenty independent tests were made.

Average	Std. Dev	Min	Max	Confidence Interval
141,874 s	16,237 s	87,37 s	160,65 s	141,874s ± 7,1

Table 5.1: Startup time

The conclusion of the average time is that the first system PoC takes to be ready in order to receive streams and treat them with QoS is due to the SDN controller instantiated in the PoC. If the SDN Controller is already instantiated, the bootstrap time will be smaller, the results with the second PoC will report that.

e) How long takes to a service be prioritized

The prioritization of one service is requested by a Hypertext Transfer Protocol (HTTP) request to the VSA. Since the PoC has all the components in the same machine, the delay that the request will take is the best case delay. The results are in the table 5.2.

Average	Std. Dev	Min	Max	Confidence Interval
4,30 ms	0,55 ms	3,33 ms	5,56 ms	4,3 ms ± 0,24

Table 5.2: How long takes to a service be prioritized

f) Percentage of packets lost without QoS

Test	Average	Std. Dev	Min	Max	Confidence Interval
1 stream simultaneously	91,66%	2,43%	83,10%	95,60%	91,66% ± 1,1%
2 streams simultaneously	70,59%	11,33%	33,8%	89,1%	70,59% ± 5%
5 streams simultaneously	68,45%	13,06%	22,9%	89,7%	68,45% ± 5,7%

Table 5.3: Percentage of packets lost without QoS

Test	Average	Std. Dev	Min	Max	Confidence Interval
1 stream simultaneously	1772,50	9,10	1721	1794	1772,5 ± 4
2 streams simultaneously	1734,68	24,66	1623	1783	1734,68 ± 11
5 streams simultaneously	1353,01	82,45	1201	1570	1353,01 ± 36

Table 5.4: Number of packets sent during one minute

Test	Average	Std. Dev	Min	Max	Confidence Interval
1 stream simultaneously	8,103 Mbits/sec	0,3371 Mbits/sec	7,25 Mbits/sec	8,72 Mbits/sec	8,103 ± 0,15 Mbits/sec
2 streams simultaneously	5,42 Mbits/sec	0,60 Mbits/sec	4,09 Mbits/sec	6,96 Mbits/sec	5,42 ± 0,26 Mbits/sec
5 streams simultaneously	301,4 Kbits/sec	47,86 Kbits/sec	185 Kbits/sec	396 Kbits/sec	301,4 ± 21 Kbits/sec

Table 5.5: iPerf Mbits/sec

Since the tests were made without quality of service and the bandwidth available for all the flows by default is at the best case 10 Mbits/sec, it is expected that for all the 3 test benchs many packets will be lost.

The measures taken shows that for all the tests more than 68% in average are lost.

With 5 streams and one packet generator running continuously the network is struggling and the bandwidth available for the iPerf session shows that, it reduces from 8,10 Mbit/s with 1 stream to 5,42 Mbits/s with 2 streams and to 301Kbit/s with 5 streams running simultaneously.

The number of packets lost during the test sessions shows the same behavior, with 5 streams drops from 1734 (2 streams) to 1353 packets sent.

The available bandwidth by default was chosen considering that 10Mbits/s is enough for one stream transmission without any other traffic.

g) Percentage of packets lost with QoS

Test	Average	Std. Dev	Min	Max	Confidence Interval
1 stream simultaneously	0%	0%	0%	0%	0% ± 0%
2 streams simultaneously	0%	0%	0%	0%	0% ± 0%
5 streams simultaneously	67,60%	15,41%	36,70%	95,40%	67,6% ± 6,8%

Table 5.6: Percentage of packets lost without QoS

Test	Average	Std. Dev	Min	Max	Confidence Interval
1 stream simultaneously	990,05	140,26	634	1267	990,05 ± 61
2 streams simultaneously	1747,175	20,19	1689	1793	1747,175 ± 8,8
5 streams simultaneously	1427,69	73,68	1265	1561	1427,69 ± 32

Table 5.7: Number of packets sent during one minute

Test	Average	Std. Dev	Min	Max	Confidence Interval
1 stream simultaneously	7,86 Mbits/sec	0,45 Mbits/sec	6,28 Mbits/sec	8,62 Mbits/sec	7,86 ± 0,2 Mbits/sec
2 streams simultaneously	8,30 Mbits/sec	0,42 Mbits/sec	7,12 Mbits/sec	8,97 Mbits/sec	8,3 ± 0,18 Mbits/sec
5 streams simultaneously	8,46 Mbits/sec	0,22 Mbits/sec	8,04 Mbits/sec	8,94 Mbits/sec	8,46 ± 0,096 Mbits/sec

Table 5.8: iPerf Mbits/sec

With the quality of service enabled it is expected that with one and two streams the percentage of packets lost in a stream transmission during one minute is 0%. However, with 5 streams simultaneously it is expected that for each stream some packets are lost, since the available bandwidth link was measured at the best case 50 Mbits/sec.

The results of percentage of packets lost without QoS shows that with 1 and 2 streams simultaneously the percentage of packets lost are 0%, but with 5 streams the average of dropped packets increases to 67%, dispite that is less than the previous 68% of packets lost without quality of service. The test with 5 streams simultaneously with QoS was conditioned by the available physical bandwidth.

h) Conclusions

Concluding, the overall results shows that with QoS policy and enough bandwidth available the streams can be transmitted without packet lost depending on the number of flows. As expected only with quality of service the iPerf bandwidth session stands at 10 Mbits/sec.

Without quality of service the packets lost are high due to the packet generator session that struggles the network and causes packets losts.

5.2.4 Second phase results

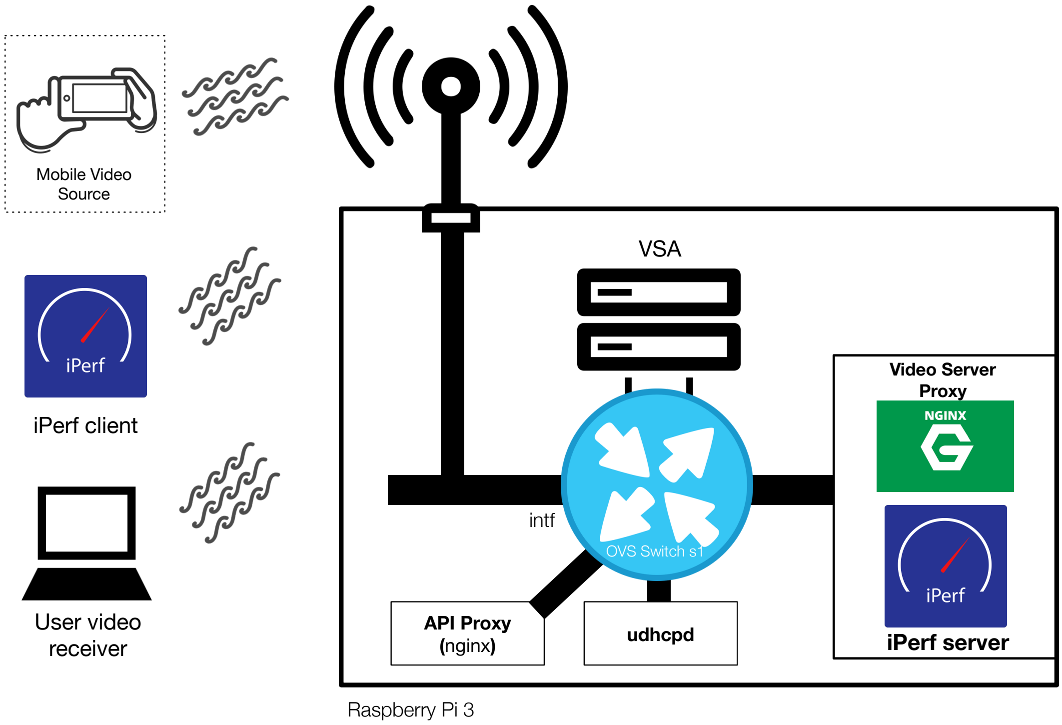

Figure 5.2: Second phase PoC

The test bench makes use of the PoC scenario deployed for the second phase depicted in the Figure 5.2 using a Raspberry. Some minor changes were made in order to get the metrics requested.

a) Characteristics of the test bench stream

- **Streaming video resolution:** 852x532
- **text**
- **FPS:** 30/1
- **Bit-rate:** 2000 Kbps
- **Buffer size:** 2000
- **Codification:** Software (x264)
- **Streaming protocol:** RTMP
- **Standard network bandwidth:** 10Mbps
- **Streaming time:** 1 minute
- Streaming from the sender to the rendez-vous point
- Only one client watches one stream
- **Downscale filter:** Bicubic
- **Format:** NV12
- **Downscale filter:** Bicubic

b) PoC deploy

The PoC scenario has been deployed in a Raspberry Pi 3 with the following configurations:

- **Main memory:** 1024 MB
- **Processors:** 1.2GHz 64-bit quad-core ARMv8 CPU
- **Interface:** 802.11n Wireless LAN

c) Test bench process order

- Instantiation of the system
- Instantiate N programs that uploads to the service provider (N = number of client uploading streams)
- Start the traffic generator
- Start the uploading stream
- Count one minute
- Collect the results
- Run for each test bench 20 independent tests

d) PoC system startup results

Like in the first phase PoC, the system startup time is independent of the clients that upload streams number to the service provider. To get the following results in table 5.9.

Average	Std. Dev	Min	Max	Confidence Interval
3,62s	0,08s	3,14s	3,76s	3,62 ± 0,035s

Table 5.9: Number of packets lost by flow and delay that the stream takes to start

The average time is lower than the first phase and that is because the controller was removed from the second phase PoC.

e) How long takes to a service be prioritized

As described before, now the action of prioritizing one service is taken by a HTTP request to the VSA API. The application in the Android mobile phone is responsible for sending the request for the QoS since the monitor application was removed from the architecture. Results are in the table 5.10.

Average	Std. Dev	Min	Max	Confidence Interval
3,75 ms	0,84 ms	3,03 ms	7,24 ms	3,75 ± 0,37 ms

Table 5.10: How long takes to a service be prioritized

f) Percentage of packets lost without QoS

Since the tests were made without quality of service and the bandwidth available for all the flows by default is at the best case 10 Mbits/sec, so is expected that for all the 3 test bench many packets are lost.

Test	Average	Std. Dev	Min	Max	Confidence Interval
1 stream simultaneously	52,29%	16,32%	22,9%	74,6%	52,29% ± 7,2%
2 streams simultaneously	27,91%	6,17%	17,1%	39,8%	27,91% ± 2,7%
5 streams simultaneously	49,82%	13,61%	23,2%	84,2%	49,82% ± 6%

Table 5.11: Percentage of packets lost without QoS

Test	Average	Std. Dev	Min	Max	Confidence Interval
1 stream simultaneously	1780,9	9,91	1746	1808	1780,9 ± 4,3
2 streams simultaneously	1789,25	19,11	1713	1819	1789,25 ± 8,4
5 streams simultaneously	1736,48	18,30	1702	1778	1736,48 ± 8

Table 5.12: Number of packets sent during one minute

Test	Average	Std. Dev	Min	Max	Confidence Interval
1 stream simultaneously	6,84 Mbits/sec	0,43 Mbits/sec	5,59 Mbits/sec	7,65 Mbits/sec	6,84 ± 0,19 Mbits/sec
2 streams simultaneously	6,33 Mbits/sec	0,24 Mbits/sec	5,96 Mbits/sec	6,91 Mbits/sec	6,33 ± 0,11 Mbits/sec
5 streams simultaneously	4,29 Mbits/sec	0,09 Mbits/sec	4,12 Mbits/sec	4,52 Mbits/sec	4,29 ± 0,039 Mbits/sec

Table 5.13: iPerf Mbits/sec

The measurement taken show that for all the tests more than 25% of packet in average are lost. Less than the previous PoC due the different test bench characteristics.

With 5 streams and one packet generator running continuously the network is struggling and the bandwidth available for the iPerf season shows that, it downs from 6,84 Mbit/s with 1 stream to 6,33 Mbits/s with 2 streams and to 4,29Mbits/s with 5 streams running simultaneously.

The available bandwidth by default was chosen considering that 10Mbps is enough for one stream transmission without any other traffic.

g) Percentage of packets lost with QoS

Test	Average	Std. Dev	Min	Max	Confidence Interval
1 stream simultaneously	0%	0%	0%	0%	0% ± 0%
2 streams simultaneously	0%	0%	0%	0%	0% ± 0%
5 streams simultaneously	12,62%	8,94%	0,00%	45,30%	12,62% ± 3,9%

Table 5.14: Percentage of packets lost without QoS

Test	Average	Std. Dev	Min	Max	Confidence Interval
1 stream simultaneously	1746,35	25,42	1702	1798	1746,35 ± 11
2 streams simultaneously	1774,43	32,13	1706	1832	1774,43 ± 14
5 streams simultaneously	1743,41	17,26	1710	1778	1743,41 ± 7,6

Table 5.15: Number of packets sent during one minute

Test	Average	Std. Dev	Min	Max	Confidence Interval
1 stream simultaneously	8,24 Mbits/sec	0,28 Mbits/sec	7,71 Mbits/sec	8,72 Mbits/sec	8,24 ± 0,12 Mbits/sec
2 streams simultaneously	9,46 Mbits/sec	0,24 Mbits/sec	8,52 Mbits/sec	9,91 Mbits/sec	9,46 ± 0,11 Mbits/sec
5 streams simultaneously	6,07 Mbits/sec	0,16 Mbits/sec	5,62 Mbits/sec	6,36 Mbits/sec	6,07 ± 0,07 Mbits/se

Table 5.16: iPerf Mbits/sec

With quality of service enabled it is expected that with one and two streams the percentage of packets lost in a stream transmission during one minute is 0%. However, with 5 streams simultaneously is expected that for each stream some packets are lost, since the available bandwidth link was measured at the best case 60 Mbits/sec.

The results of percentage of packets lost without QoS shows that with 1 and 2 streams simultaneously the percentage of packets lost are 0%, but with 5 streams the average of dropped packets increases to 12%. Although is less than the previous 49% of packets lost without quality of service. The test with 5 streams simultaneously with QoS was conditioned by the available physical bandwidth.

h) Conclusions

Concluding the overall results, like in the first PoC, results show that with QoS policy and enough bandwidth available the streams can be transmitted without packet loss. As expected only with quality of service, the iPerf bandwidth session stands closer to 10 Mbits/sec.

Without quality of service the packet loss is high due to the packet generator session that struggles the network and causes packets losts. All the packets transmitted by the packet generator are TCP packets simulating a TCP video session and by the RTMP streaming session are TCP too.

Since the second PoC relies in the Wi-Fi of the Raspberry Pi 3 as the available physical bandwidth, the results were different from the first PoC. Nonetheless, the conclusions are the same.

CHAPTER 6

Conclusions and Future Work

This work presented a solution to prioritize uplink flows. Network providers can benefit from this by making an interface for service providers prioritize flows inside their network.

Given the increase interest in live video sharing, this kind of solution can be used to help minimize the problems with the delayed networks. Although this work provides only prioritization, it was been inserted in the VDSNet project that intends to optimize the uplink live stream flows. And with the participation in that project it was possible to create a modular architecture and network.

This work makes use of software-defined networks, which despite being cutting edge technology, may require adaptations to be usable in an operator is legacy network. However, the architecture followed, idea and concept will be easily implemented in an architecture of a traditional or SDN-based operator.

The development of a PoC allowed to validate the solution, although it is not a production version, it can exemplify several problems and advantages that a network based on this solution has. The biggest problem persists in the physical capacity of the link that is always limited at some number but it is out of the scope of this work.

The obtained results and compartmental behavior are the expected.

6.1 Future work

6.1.1 Deployment in a network operator architecture

The validation by an operator requires more precise results in terms of flows supported by this system and deploying into a major network requires some concept modification in order to scale.

As described in Figure 6.1 the deployment scenario is detailed as:

1. The monitor application can be in the access nodes (Wi-Fi access point or e-node B) and can communicate with the VSA
2. The VSA can have multiple operation methods as described in chapter 3.1.3. That possibility gives to the VSA classification and business-to-service decision power.

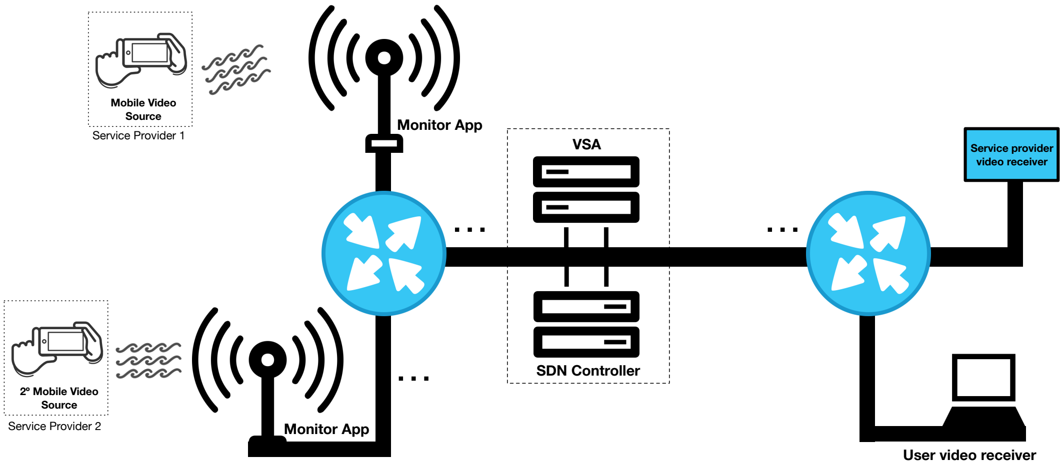

Figure 6.1: Conceptual architecture

3. SDN Controller it is not vendor locked, can be any Controller that implements the required operations by the VSA
4. OVS Switch can and should be replaced by a Operator-SDN Architecture
5. The video server proxy it is replaced by the service provider architecture, and should be notified to the VSA how the service provider can operate

6.1.2 Mechanisms to adapt the transmission at the mobile device

Creating a module on the user is terminal would allow the selection of codecs and receive settings. It would also help in scheduling at the uplink and with that can optimize the uplink through scheduling in the radio physical link.

CHAPTER 7

Appendix

This chapter is intended to bring together all the settings, examples, and commands that are useful for explaining how all the work was done. In order to register certain ways of thinking, developing and obtaining the final product, it is very necessary that it be preserved in a chapter.

Placing between all the writing would make the text difficult to read.

7.1 Configurations

7.1.1 Examples

Dpctl example

Query for flow stats:

```
# dpctl unix:/var/run/s1.sock stats-flow
```

```
SENDING:
stat_req{type="flow", flags="0x0", table="all", oport="any", ogrp="any", cookie=0x0",
mask=0x0", match=oxm{all match}}

RECEIVED:
stat_repl{type="flow", flags="0x0", stats=[{table="0", match="oxm{in_port="2",  eth_type=0x"800"}",
dur_s="1", dur_ns="342000", prio="2048", idle_to="30", hard_to="300",
 cookie="0x0", pkt_cnt="0", byte_cnt="0",
insts=[apply{acts=[out{port="1"}]}]}]}
```

7.1.2 Installations

OpenDayLight installation

Update the package lists from the repositories to get information from the newest versions of packages and their dependencies.

```
# apt-get update
```

Since OpenDayLight has been built in top of Java, it is needed to install Java and the maven (dependencies manager) and the tool wget for the next steps.

```
# apt-get install -y openjdk-7-jdk openjdk-7-jre maven wget
```

Download the binaries from the OpenDayLight website.

```
# wget https://nexus.opendaylight.org/content/repositories/opendaylight.release/org/
opendaylight/integration/distribution-karaf/0.4.4- Beryllium-SR4/distribution-karaf-0.4.4-
Beryllium-SR4.tar.gz
```

Uncompress the file and rename the folder name.

```
# tar xvzf distribution-karaf-0.4.4-Beryllium-SR4.tar.gz && rm *.tar.gz && mv
distribution-karaf-0.4.4-Beryllium-SR4 odl
```

Now it is important to know what features should be enabled for the OpenDayLight installation[1,2].

- **odl-restconf**: Enables REST API access to the MD-SAL[3] including the data store
- **odl-l2switch** and **odl-l2switch-switch-ui**: Provides L2 (Ethernet) forwarding across connected OpenFlow switches and support for host tracking
- **odl-mdsal-apidocs**: MD-SAL API Documentation
- **odl-dlux-core** and **odl-dlux-all**: installs the OpenDaylight Web Interface (DLUX) and shows information about the OpenFlow data and L2 Switch components such as: network, flow statistics, host locations.

In order to enable these features the file */odl/etc/org.apache.karaf.features.cfg* should have the following content:

```
featuresRepositories = mvn:org.apache.karaf.features/standard/3.0.3/xml/features,
mvn:org.apache.karaf.features/enterprise/3.0.3/xml/features,
mvn:org.ops4j.pax.web/pax-web-features/3.1.4/xml/features,
mvn:org.apache.karaf.features/spring/3.0.3/xml/features,
mvn:org.opendaylight.integration/features-integration-index/0.4.4-Beryllium-SR4/
xml/features

featuresBoot=config,standard,region,package,kar,ssh,management,odl-restconf,
odl-l2switch,odl-l2switch-switch-ui,odl-mdsal-apidocs,odl-dlux-core,odl-dlux-all

featuresBootAsynchronous=false
```

Now the OpenDayLight installation it is ready to go.

```
# ./odl/bin/karaf -of13
```

OpenDayLight Dockerfile

Dockerfile

```
FROM ubuntu:14.04
RUN apt-get update
RUN apt-get install -y openjdk-7-jdk openjdk-7-jre maven wget
```

[1] https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL

[2] https://www.opendaylight.org/sites/opendaylight/files/bk-install-guide-20150831.pdf

[3] Model-Driven SAL (MD-SAL) provides common and generic support to application and plugin developers

```
RUN wget https://nexus.opendaylight.org/content/repositories/opendaylight.
release/org/opendaylight/integration/distribution-karaf/0.4.4-Beryllium-SR4
/distribution-karaf-0.4.4-Beryllium-SR4.tar.gz
RUN tar xvzf distribution-karaf-0.4.4-Beryllium-SR4.tar.gz
RUN rm distribution-karaf-0.4.4-Beryllium-SR4.tar.gz
RUN mv distribution-karaf-0.4.4-Beryllium-SR4 odl

COPY org.apache.karaf.features.cfg /odl/etc/org.apache.karaf.features.cfg

EXPOSE 1088 1830 2400 5666 6633 6653 7800 8000 8181 8383 12001
WORKDIR /odl/bin
CMD ./karaf -of13
```

org.apache.karaf.features.cfg

```
featuresRepositories = mvn:org.apache.karaf.features/standard/3.0.3/xml/
features,mvn:org.apache.karaf.features/enterprise/3.0.3/xml/features,
mvn:org.ops4j.pax.web/pax-web-features/3.1.4/xml/features,mvn:org.
apache.karaf.features/spring/3.0.3/xml/features,mvn:org.opendaylight.
integration/features-integration-index/0.4.4-Beryllium-SR4/xml/features
featuresBoot=config,standard,region,package,kar,ssh,management,odl-restconf,odl-l2switch,
odl-l2switch-switch-ui,odl-mdsal-apidocs,odl-dlux-core,odl-dlux-all
featuresBootAsynchronous=false
```

7.1.3 First phase PoC

Modifications in the Mininet VM

After downloading the Mininet VM (Mininet 2.2.1 on Ubuntu 14.04 LTS), two interfaces has been added bridging into the physical network. It can be added only one interface but, one must have no IP address at all.

After that, remove the NAT interface that is not useful for the PoC. Last, install guest additions and configure a share folder to share the work files or send them using scp[4].

RTMP NGINX server with iPerf

Dockerfile

```
FROM ubuntu:trusty
RUN apt-get update
RUN apt-get install -y build-essential libpcre3 libpcre3-dev libssl-dev unzip wget
RUN mkdir nginx
RUN cd nginx
RUN wget http://nginx.org/download/nginx-1.8.1.tar.gz
RUN tar -zxvf nginx-1.8.1.tar.gz
RUN wget https://github.com/arut/nginx-rtmp-module/archive/master.zip
RUN unzip master.zip
WORKDIR nginx-1.8.1
RUN ./configure --with-http_ssl_module
--add-module=../nginx-rtmp-module-master
RUN make
RUN make install
```

[4] https://linuxacademy.com/blog/linux/ssh-and-scp-howto-tips-tricks/

```
# /usr/local/nginx/sbin/nginx -s stop
COPY nginx.conf /usr/local/nginx/conf/nginx.conf
RUN mkdir /usr/local/nginx/rtmp
RUN /usr/local/nginx/sbin/nginx
# iperf
# install binary and remove cache
RUN apt-get install -y software-properties-common python-software-properties
screen
RUN add-apt-repository "ppa:patrickdk/general-lucid"
RUN apt-get update && apt-get install -y iperf3
EXPOSE 1935 8080 5201
CMD ["/bin/bash"]
```

nginx.conf

```
worker_processes 1;

events {
worker_connections 1024;
}

http {
include       mime.types;
default_type  application/octet-stream;

sendfile        on;
keepalive_timeout  65;


server {
listen       8080;
server_name  localhost;

location /hls {
types {
application/vnd.apple.mpegurl m3u8;
video/mp2t ts;
}
root /tmp;
add_header Cache-Control no-cache;
}

location /on_publish {
return 201;
}

location /stat {
rtmp_stat all;
rtmp_stat_stylesheet stat.xsl;
}
```

```
location /stat.xsl {
root /opt/nginx/conf/stat.xsl;
}

location /control {
rtmp_control all;
}

error_page   500 502 503 504  /50x.html;
location = /50x.html {
root   html;
}

}

}

rtmp {
server {
listen 1935;
chunk_size 4096;
application live {
live on;
record off;
on_publish http://localhost:8080/on_publish;
hls on;
hls_path /tmp/hls;
}
}
}
```

Containernet installation

Install the ansible, git and aptitude. Ansible is a provisioning tool that helps to install the required tools for a specific tool. Git is a version control system and Apititude is a look-a-like "apt" tool with more features.

```
# sudo apt-get install -y ansible git aptitude
```

Then there are some libraries that are required:

```
# sudo apt-get install -y python-dev libxml2-dev libxslt-dev
# sudo apt-get install -y libffi-dev
```

Edit the */etc/ansible/hosts* file to specify where the containernet must be installed. Open and then paste: "localhost ansible_connection=local"

```
# sudo vim /etc/ansible/hosts
```

Then, install some python required packages:

```
# pip install pytest
# pip install -upgrade setuptools
# pip install -upgrade urllib3
```

pip install -e 'git+https://github.com/shin-/compose.git@66f4a795a2ded1a26b6cf84 74edb423727dd585d#egg=docker-compose'

Create a folder to place the source of Containernet:

```
# mkdir /home/containernet
# cd /home/containernet
```

Clone the source of Containernet:

```
# git clone https://github.com/containernet/containernet.git .
```

And install it using ansible:

```
# cd ansible
# ansible-playbook install.yml
```

7.1.4 Second phase PoC

Hostapd

There are some important steps that have to be described how the hostapd has been configured using a Raspberry Pi 3.

First, install the hostapd:

```
# sudo apt-get install -y hostapd
```

Change the */etc/init.d/hostapd* file environment vars to:

```
PATH=/sbin:/bin:/usr/sbin:/usr/bin
DAEMON\_SBIN=/usr/sbin/hostapd
DAEMON\_DEFS=/etc/default/hostapd
DAEMON\_CONF=/etc/hostapd/hostapd.conf
```

Then, change the */etc/default/hostapd* file environment vars to:

```
DAEMON_CONF="/etc/hostapd/hostapd.conf"
```

In the interfaces file */etc/network/interfaces*, the interface can have a static IP only for first instance run.

```
auto wlan0
allow-hotplug wlan0

iface wlan0 inet static
address 192.168.10.1
netmask 255.255.255.0
```

The hostapd file */etc/hostapd/hostapd.conf* will have the Wi-Fi configurations. The network must have no password:

```
interface=wlan0
driver=nl80211
ssid=VDSNET
hw_mode=g
channel=1
```

Next, activate the forwarding capabilities:

```
# sudo sh -c "echo 1 > /proc/sys/net/ipv4/ip_forward"
```

If want to test the configurations with debug:

```
# sudo hostapd -dd /etc/hostapd/hostapd.conf
```

Containernet

First, let is install vim and tmux. Tmux[5] is a terminal multiplexer that provide a API to interact with the terminal window and panes. Using the python library this tool it will be useful in order to monitor the auto initialization of the work in Raspberry Pi.

```
# sudo apt-get install -y vim tmux
```

After that, let is install docker, it has an installation script for Raspbian[6] (Raspberry Pi operating system) fully supported by Docker.

```
# curl -sSL https://get.docker.com | sh
```

Let is add the user *pi* to the docker group.

```
# sudo usermod -aG docker pi
```

There are more libraries needed to the containernet installation, for example the tool Ansible:

```
# sudo apt-get install -y ansible git aptitude
# sudo apt-get install -y python-dev libxml2-dev libxslt-dev
# sudo apt-get install -y libffi-dev
```

Then, let is add the localhost to the ansible hosts by adding the following line to the file */etc/ansible/hosts*:

```
localhost ansible_connection=local
```

Some python tools are required, such as "pip[7]" package installer.

```
# sudo apt-get install -y python-pip
# sudo apt-get install -y pip
# wget https://bootstrap.pypa.io/get-pip.py
# sudo python get-pip.py
```

Now, let is modify the Containernet project in order to make it work with Raspberry.

```
# mkdir containernet
# cd containernet
# git clone https://github.com/containernet/containernet.git .
```

Remove the four lines after the line 71 in the file "util/install.sh":

```
#if ! echo $DIST | egrep 'Ubuntu|Debian|Fedora|RedHatEnterpriseServer'; then
#    echo "Install.sh currently only supports Ubuntu,
Debian, RedHat and Fedora."
#    exit 1
#fi
```

And in the file *ansible/install.yml* remove the docker lines and then install Containernet project:

```
# sudo ansible-playbook install.yml
```

[5]https://github.com/tmux/tmux/wiki
[6]https://www.raspberrypi.org/downloads/raspbian/
[7]https://pip.pypa.io/en/stable/installing/

Last but not least, some python packages must be installed:

```
# sudo pip install pytest
# sudo pip install -upgrade setuptools
# sudo pip install colorama
# sudo pip install netifaces
# sudo pip install libtmux
# sudo pip install flask
# sudo pip install flask_restful
```

After the packages and containernet project were installed, let is comment the file "/home-/pi/containernet/mininet/node.py" in the line 703.

```
self.resources = dict(
# cpu_quota=defaults['cpu_quota'],
cpu_period=defaults['cpu_period'],
cpu_shares=defaults['cpu_shares'],
cpuset_cpus=defaults['cpuset_cpus'],
mem_limit=defaults['mem_limit'],
memswap_limit=defaults['memswap_limit']
)
```

Docker images

DHCP docker image

Dockerfile

```
FROM resin/rpi-raspbian
RUN apt-get update
RUN apt-get install -y udhcpd net-tools iputils-ping
COPY udhcpd.conf /etc/udhcpd.conf
COPY udhcpd /etc/default/udhcpd
RUN service udhcpd start
EXPOSE 67
EXPOSE 67/udp
EXPOSE 68
EXPOSE 68/udp
CMD ["/bin/bash"]
```

udhcp

```
# Comment the following line to enable
#DHCPD_ENABLED="no"
DHCPD_OPTS="-S"
```

udhcp.conf

```
start                192.168.10.100        #default: 192.168.0.20
end                    192.168.10.199        #default: 192.168.0.254

interface        d4-eth0        #default: eth0

/var/lib/misc/udhcpd.leases
```

```
/var/run/udhcpd.pid

option       subnet       255.255.255.0
opt       router       192.168.10.1
servers for a total of 3
option       domain       local
option       lease       864000              # 10 days of seconds
```

NGINX API Proxy image

Dockerfile

```
FROM resin/rpi-raspbian
RUN apt-get update
RUN apt-get install -y build-essential libpcre3 libpcre3-dev libssl-dev unzip wget net-tools
RUN mkdir nginx
RUN cd nginx
RUN wget http://nginx.org/download/nginx-1.8.1.tar.gz
RUN tar -zxvf nginx-1.8.1.tar.gz
WORKDIR nginx-1.8.1
COPY nginx.conf /nginx-1.8.1/conf/nginx.conf
RUN ./configure
RUN make
RUN make install
RUN /usr/local/nginx/sbin/nginx
RUN apt-get install -y iputils-ping
EXPOSE 80
CMD ["/bin/bash"]
```

nginx.conf

```
#user  nobody;
worker_processes  1;

#error_log  logs/error.log;
#error_log  logs/error.log  notice;
#error_log  logs/error.log  info;

#pid        logs/nginx.pid;

events {
worker_connections  1024;
}

http {
include       mime.types;
default_type  application/octet-stream;
"$http_x_forwarded_for"';

sendfile        on;
#tcp_nopush     on;
```

```
keepalive_timeout  65;

server {
listen       80;
server_name  localhost;

location / {
proxy_pass   http://172.17.0.1/;
}
}
}
```

Startup script

startup.py

```python
#!/usr/bin/python

from mininet.net import Mininet
from mininet.link import Intf
from mininet.log import setLogLevel, info
from colors import info, error, success
import json
import subprocess, signal, os
import netifaces as ni
import libtmux
import time

with open('config.json') as json_data:
d = json.load(json_data)


NETWORK = d["network"]
print("startup.py | Network: " + NETWORK)


def clean_ovs():
# clean ovs flows
cmd = "ovs-ofctl del-flows " + d["bridge"]
info(cmd)
p = subprocess.Popen(cmd, stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()

# destroy qos rules and queue in OVS
# ovs-vsctl -- destroy QoS $d["ovs_interface"] -- clear Port $d["ovs_interface"]  qos
info("ovs-vsctl -- destroy QoS "+d["ovs_interface"]+" -- clear Port "+d["ovs_interface"]+"  qos")
p = subprocess.Popen('ovs-vsctl -- destroy QoS ' + d["ovs_interface"] + ' -- clear Port '
 + d["ovs_interface"] + '  qos', stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()
```

```
# ovs-vsctl clear Port $d["ovs_interface"] qos
info("ovs-vsctl clear Port "+d["ovs_interface"]+"  qos")
p = subprocess.Popen('ovs-vsctl clear Port ' +
 d["ovs_interface"] + ' qos', stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()

# ovs-vsctl list qos
info("ovs-vsctl list qos")
p = subprocess.Popen('ovs-vsctl list qos', stdout=subprocess.PIPE,
 shell=True)
(output, err) = p.communicate()
p_status = p.wait()

if "_uuid" in output:
line = output.splitlines()[0]
uuid = line.split(":")[1].replace(" ", "")
# ovs-vsctl destroy qos
info("ovs-vsctl destroy qos " + uuid)
p = subprocess.Popen("ovs-vsctl destroy qos " + uuid, stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()

# ovs-vsctl list Queue
info("ovs-vsctl list Queue")
p = subprocess.Popen('ovs-vsctl list Queue', stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()

if "_uuid" in output:
line = output.splitlines()[0]
uuid = line.split(":")[1].replace(" ", "")
# ovs-vsctl  destroy Queue
info("ovs-vsctl destroy Queue " + uuid)
p = subprocess.Popen("ovs-vsctl destroy Queue " + uuid, stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()

success("OVS qos and queues clean done!")


def verify_network():
net = Mininet(topo=None, build=False)

info('*** Adding controller\n')
net.addController(name='c0')

info('*** Add switches\n')
s1 = net.addSwitch('s1')
Intf(d["interface"], node=s1)
```

```
info('*** Add hosts\n')
ip = NETWORK+'210'
h1 = net.addHost('h1', ip=NETWORK+'210')

try:
info('*** Add links\n')
net.addLink(h1, s1)
except Exception:
error("RTNETLINK answers: File exists")
info("Please restart the VM ...")
exit(1)

info('*** Starting network\n')
net.start()
info('*** Verify IP')
result = h1.cmd("ifconfig")
# CLI(net)
net.stop()

if ip in result:
success("IP verified")
else:
error("IP not correct, please check!")
exit(1)


def create_tmux():
# connect to tmux server
server = libtmux.Server()

try:
# attach tmux server session
session = server.new_session("vdsnet", detach=True)
# create window for the session created for vdsnet
session.new_window(attach=False, window_name="network py")
# attach window session
window = session.attached_window
# in order to create panes (for type) we must split the window
window.split_window(attach=False)
# now the panes are created, the 0 will be to vsa and the 1 will be to vdsnet

# attach tmux server session
session = server.new_session("iperf", detach=True)
# create window for the session created for vdsnet
session.new_window(attach=False, window_name="iperf session")
# attach window session
window = session.attached_window
# in order to create panes (for type) we must split the window
window.split_window(attach=False)
```

```
except libtmux.exc.TmuxSessionExists:
pass


def start_vsa():
# connect to tmux server
server = libtmux.Server()
# attach tmux server session
session = server.find_where({"session_name": "vdsnet"})
# attach window session
window = session.attached_window
# list panes and select the first element
pane = window.list_panes()[0]
# cd vdsnet and then start the network
pane.send_keys('cd vsa', enter=True)
pane.send_keys('sudo python vsa.py --web &>/dev/null', enter=True)
# pane.send_keys('exit', enter=True)
# print('\n'.join(pane.cmd('capture-pane', '-p').stdout))


def start_vdsnet():
# connect to tmux server
server = libtmux.Server()
# attach tmux server session
session = server.find_where({"session_name": "vdsnet"})
# attach window session
window = session.attached_window
# list panes and select the first element
pane = window.list_panes()[1]
# cd vdsnet and then start the network
pane.send_keys('cd vdsnet', enter=True)
pane.send_keys('sudo python vdsnet.py', enter=True)
# print('\n'.join(pane.cmd('capture-pane', '-p').stdout))


def wait_for_start():
while True:
time.sleep(1)
cmd = "service docker status"
info(cmd)
p = subprocess.Popen(cmd, stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()

if "active (running)" in output:
return


if __name__ == '__main__':
setLogLevel('info')
```

```
wait_for_start()
clean_ovs()
verify_network()
create_tmux()
start_vsa()
start_vdsnet()
success("bootsrap done!")
```

vdsnet.py

```
#!/usr/bin/python

from mininet.net import Containernet
from mininet.net import Mininet
from mininet.cli import CLI
from mininet.link import Intf
from mininet.log import setLogLevel, info
from colors import info, error, success
import json
import subprocess
import netifaces as ni

with open('config.json') as json_data:
d = json.load(json_data)


MAX_RATE = d["rate1"] if (int(d["rate1"]) > int(d["rate2"])) else d["rate2"]
NETWORK = d["network"]
print("vdsnet.py | NETWORK: " + NETWORK)


def create_queues():
info("creating queues...")
# ovs-vsctl set port $d["ovs_interface"] qos=@newqos -- --id=@newqos create qos type=linux-htb
other-config:max ..
cmd = "ovs-vsctl set port " + d["ovs_interface"] + " qos=@newqos -- --id=@newqos create qos
 type=linux-htb " \
" other-config:max-rate=" + MAX_RATE + " queues=1=@q1,2=@q2 -- --id=@q1 create queue
other-config:min-rate=" + d["rate1"] + \
" other-config:max-rate=" + d["rate1"] + " -- --id=@q2 create queue other-config:min-rate=" + d["rate2"]
+ " other-config:max-rate=" + d["rate2"]

info(cmd)
p = subprocess.Popen(cmd, stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()


def clean_ovs():
# clean ovs flows
cmd = "ovs-ofctl del-flows " + d["bridge"]
```

```
info(cmd)
p = subprocess.Popen(cmd, stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()

# destroy qos rules and queue in OVS
# ovs-vsctl -- destroy QoS $d["ovs_interface"] -- clear Port $d["ovs_interface"]  qos
info("ovs-vsctl -- destroy QoS "+d["ovs_interface"]+" -- clear Port "+d["ovs_interface"]+"  qos")
p = subprocess.Popen('ovs-vsctl -- destroy QoS ' + d["ovs_interface"] + ' -- clear Port ' +
d["ovs_interface"] + '  qos', stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()

# ovs-vsctl clear Port $d["ovs_interface"] qos
info("ovs-vsctl clear Port "+d["ovs_interface"]+"  qos")
p = subprocess.Popen('ovs-vsctl clear Port ' +
d["ovs_interface"] + ' qos', stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()

# ovs-vsctl list qos
info("ovs-vsctl list qos")
p = subprocess.Popen('ovs-vsctl list qos', stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()

if "_uuid" in output:
line = output.splitlines()[0]
uuid = line.split(":")[1].replace(" ", "")
# ovs-vsctl destroy qos
info("ovs-vsctl destroy qos " + uuid)
p = subprocess.Popen("ovs-vsctl destroy qos " + uuid, stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()

# ovs-vsctl list Queue
info("ovs-vsctl list Queue")
p = subprocess.Popen('ovs-vsctl list Queue', stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()

if "_uuid" in output:
line = output.splitlines()[0]
uuid = line.split(":")[1].replace(" ", "")
# ovs-vsctl  destroy Queue
info("ovs-vsctl destroy Queue " + uuid)
p = subprocess.Popen("ovs-vsctl destroy Queue " + uuid, stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()
```

```
success("OVS qos and queues clean done!")


def verify_network():
net = Mininet(topo=None, build=False)

info('*** Adding controller\n')
net.addController(name='c0')

info('*** Add switches\n')
s1 = net.addSwitch('s1')
Intf(d["interface"], node=s1)

info('*** Add hosts\n')

IP_NETWORK = NETWORK + "1"
h1 = net.addHost('h1', ip=IP_NETWORK)

try:
info('*** Add links\n')
net.addLink(h1, s1)
except Exception:
error("RTNETLINK answers: File exists")
info("Please restart the VM ...")
exit(1)

info('*** Starting network\n')
net.start()
info('*** Verify IP')
result = h1.cmd("ifconfig")
# CLI(net)
net.stop()

if IP_NETWORK in result:
success("IP verified")
else:
error("IP not correct, please check!")
exit(1)


def verify_docker():
# docker rm -f mn.d1
info("docker rm -f mn.d1")
p = subprocess.Popen('docker rm -f mn.d1', stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()

# docker rm -f mn.d2
info("docker rm -f mn.d2")
p = subprocess.Popen('docker rm -f mn.d2', stdout=subprocess.PIPE, shell=True)
```

```
(output, err) = p.communicate()
p_status = p.wait()

# docker rm -f mn.d3
info("docker rm -f mn.d3")
p = subprocess.Popen('docker rm -f mn.d3', stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()

# docker rm -f mn.d4
info("docker rm -f mn.d4")
p = subprocess.Popen('docker rm -f mn.d4', stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()

# iperf image
info("verify iperf image")
p = subprocess.Popen('docker images', stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
p_status = p.wait()

if "vdsnet/iperf" not in output:
error("iperf not in docker images")
error("build the image first")
exit(1)
else:
info("iperf image OK!")

# nginx rtmp image
if "vdsnet/rtmp-server" not in output:
error("rtmp server not in docker images")
error("build the image first")
exit(1)
else:
info("rtmp server image OK!")

# nginx api proxy image
if "vdsnet/api-proxy" not in output:
error("vdsnet/api-proxy not in docker images")
error("build the image first")
exit(1)
else:
info("vdsnet/api-proxy image OK!")

# dhcp image
if "vdsnet/dhcp" not in output:
error("vdsnet/dhcp not in docker images")
error("build the image first")
exit(1)
else:
```

```
info("vdsnet/dhcp image OK!")


def vdsnet_network():
net = Containernet(topo=None, build=False)

info('*** Adding controller\n')
net.addController(name='c0')

info('*** Add switches\n')
s1 = net.addSwitch('s1')
Intf(d["interface"], node=s1)

info('*** Add hosts\n')
h1 = net.addHost('h1', ip=NETWORK+'210')

info('*** Adding docker containers\n')
d1 = net.addDocker('d1', ip=NETWORK+'200', dimage="vdsnet/rtmp-server")
d2 = net.addDocker('d2', ip=NETWORK+'220', dimage="vdsnet/iperf")  # iperf3 -c 192.168.10.200
d3 = net.addDocker('d3', ip=NETWORK+'250', dimage="vdsnet/api-proxy")
d4 = net.addDocker('d4', ip=NETWORK+'230', dimage="vdsnet/dhcp")

info('*** Add links\n')
net.addLink(h1, s1)
net.addLink(d1, s1)
net.addLink(d2, s1)
net.addLink(d3, s1)
net.addLink(d4, s1)

info('*** Starting network\n')
net.start()

d1.cmdPrint('/usr/local/nginx/sbin/nginx')
d1.cmdPrint('iperf3 -s -V -D')
d3.cmdPrint('/usr/local/nginx/sbin/nginx')
d4.cmdPrint('service udhcpd start')

create_queues()
standard_flow()

CLI(net)

net.stop()


def standard_flow():
# add standard flow via command line
cmd = "ovs-ofctl add-flow " + d["bridge"] + " priority=50,tcp,nw_dst="+NETWORK+"200/32,actions=enqueue:3:1"
info(cmd)
p = subprocess.Popen(cmd, stdout=subprocess.PIPE, shell=True)
```

```
(output, err) = p.communicate()
p_status = p.wait()


if __name__ == '__main__':
setLogLevel('info')
clean_ovs()
verify_network()
verify_docker()
vdsnet_network()
```

References

[1] OpenVZ. (). Introduction to virtualization. O. .-.-. V. containers, Ed., [Online]. Available: https://wiki.openvz.org/Introduction_to_virtualization.

[2] S. P. Conroy. (2010). History of virtualization. E. VM, Ed., [Online]. Available: http://www.everythingvm.com/content/history-virtualization.

[3] N. M. K. Chowdhury and R. Boutaba, "A Survey of Network Virtualization", David R. Cheriton School of Computer Science, University of Waterloo, TS, 2008.

[4] VMware. (2005). Virtualization: Architectural considerations and other evaluation criteria. V. Inc., Ed., [Online]. Available: https://www.vmware.com/pdf/virtualization_considerations.pdf.

[5] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization", *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, p. 275, 2007. DOI: 10.1145/1272998.1273025. [Online]. Available: https://pdfs.semanticscholar.org/bf36/1962667e2cf7d8c2fde3186012cc8df87cb2.pdf.

[6] *What is docker*, Apr. 2017. [Online]. Available: https://www.docker.com/what-docker.

[7] *What is a container*, May 2017. [Online]. Available: https://www.docker.com/what-container.

[8] "Welcome to ieee nfv-sdn 2016", *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2016. DOI: 10.1109/nfv-sdn.2016.7919462.

[9] J. Esch, "Prolog to, "software-defined networking: A comprehensive survey"", *Proceedings of the IEEE*, vol. 103, no. 1, pp. 10–13, 2015. DOI: 10.1109/jproc.2014.2374752.

[10] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer, and O. Koufopavlou, "Software-defined networking (sdn): Layers and architecture terminology", RFC Editor, RFC 7426, Jan. 2015, http://www.rfc-editor.org/rfc/rfc7426.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7426.txt.

[11] *Ipv6 flow label specification*. [Online]. Available: https://tools.ietf.org/html/rfc3697.

[12] *Instructions and actions*. [Online]. Available: http://flowgrammable.org/sdn/openflow/actions/#tab_ofp_1_1.

[13] Openvswitch, *Openvswitch/ovs*. [Online]. Available: https://github.com/openvswitch/ovs/blob/master/Documentation/intro/why-ovs.rst.

[14] P. Phaal, S. Panchen, and N. McKee, "Inmon corporation is sflow: A method for monitoring traffic in switched and routed networks", RFC Editor, RFC 3176, Sep. 2001, http://www.rfc-editor.org/rfc/rfc3176.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3176.txt.

[15] B. Claise, "Cisco systems netflow services export version 9", RFC Editor, RFC 3954, Oct. 2004, http://www.rfc-editor.org/rfc/rfc3954.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3954.txt.

[16] L. Yan and N. Mckeown, "Learning networking by reproducing research results", *ACM SIGCOMM Computer Communication Review*, vol. 47, no. 2, pp. 19–26, Feb. 2017. DOI: 10.1145/3089262.3089266.

[17] M. Peuster, H. Karl, and S. V. Rossem, “Medicine: Rapid prototyping of production-ready network services in multi-pop environments”, *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2016. DOI: 10.1109/nfv-sdn.2016.7919490.

[18] Containernet, *Containernet/containernet*, May 2017. [Online]. Available: https://github.com/containernet/containernet.

[19] H. Nam, D. Calin, and H. Schulzrinne, “Intelligent content delivery over wireless via sdn”, *2015 IEEE Wireless Communications and Networking Conference (WCNC)*, 2015. DOI: 10.1109/wcnc.2015.7127806.

[20] R. Haw, C. S. Hong, and S. Lee, “An efficient content delivery framework for sdn based lte network”, *Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication - ICUIMC 14*, 2014. DOI: 10.1145/2557977.2558087.

[21] V. Yazıcı, U. Kozat, and M. Sunay, “A new control plane for 5g network architecture with a case study on unified handoff, mobility, and routing management”, *IEEE Communications Magazine*, vol. 52, no. 11, pp. 76–85, 2014. DOI: 10.1109/mcom.2014.6957146.

[22] *All about wired and wireless technology.* [Online]. Available: http://4g-lte-world.blogspot.pt/2014/05/basics-of-scheduling-in-lte.html.

[23] *All about wired and wireless technology.* [Online]. Available: http://4g-lte-world.blogspot.fr/2013/01/quality-of-service-qos-in-lte.html.

[24] H. Schulzrinne, A. Rao, and R. Lanphier, “Real time streaming protocol (rtsp)”, RFC Editor, RFC 2326, Apr. 1998, http://www.rfc-editor.org/rfc/rfc2326.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2326.txt.

[25] M. Thornburgh, “Adobe is rtmfp profile for flash communication”, RFC Editor, RFC 7425, Dec. 2014.

[26] D. Athow, *Why network operators must embrace innovation – and otts – to survive*, Feb. 2015. [Online]. Available: http://www.techradar.com/news/phone-and-communications/why-network-operators-must-embrace-innovation-and-otts-to-survive-1286103.